MASTER'S THESIS 2021

Unsupervised Prediction of Software Faults using Change Metrics

Oskar Holmqvist, Elias Tedenvall

ISSN 1650-2884
LU-CS-EX: 2021-31

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2021-31

Unsupervised Prediction of Software Faults using Change Metrics

Oövervakad felprediktion av mjukvara med förändringsfaktorer

Oskar Holmqvist, Elias Tedenvall

Unsupervised Prediction of Software Faults using Change Metrics

Oskar Holmqvist
os4226ho-s@student.lu.se

Elias Tedenvall
el2725te-s@student.lu.se

August 10, 2021

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

Supervisors: Pierre Nugues, pierre.nugues@cs.lth.se
Henrik Forsberg, henrik.forsberg@axis.com
Kenneth Nilsson, nilsson.kenneth@gmail.com

Examiner: Martin Höst, martin.host@cs.lth.se

Abstract

Software quality assurance is of great importance in all software projects. Since it is a time consuming process, it has seen a lot of development during the last years with machine learning as a tool. This thesis explores how software fault prediction can be used to speed up the process of software quality assurance. We developed unsupervised machine learning models based on the different clustering techniques K-Means, DBSCAN, Agglomerative, Gaussian Mixture Models and Fuzzy-C-Means. Precision and recall was calculated for each clustering technique and we found that Fuzzy-C-Means performed the best. To develop the model, we used change metrics in Git as features in order to speed up the process of feature extraction. We validated the models with the *Technical Debt Dataset*, which is a labeled dataset.

For Fuzzy-C-Means, the AUC score was 0.68. With a distribution of 10% fault-inducing commits in the dataset, the model achieved a precision of 0.1841 and a recall of 0.7237. With 1% fault-inducing commits in the dataset, the precision was 0.0192 and the recall 0.7464. We also validated the model against issues in Jira that reported faults. We discovered a correlation between commits that were clustered as fault-prone and the Jira issues from the same project. Our results show a promising future for unsupervised fault-proneness prediction.

Keywords: Quality Assurance, Software Fault Prediction, Unsupervised Machine Learning, Clustering, Fuzzy-C-Means, The Technical Debt Dataset, Jira

Acknowledgements

We would like thank Pierre Nugues, our supervisor at LTH, for his great guidance and support throughout the whole thesis process. He has provided us with both great machine learning strategies as well as good advice regarding the writing of this report.

We would also like to thank our supervisors at Axis, Henrik Forsberg and Kenneth Nilsson, for their great support and input regarding Axis related information.

Lastly we would like to thank Axis the company as a whole for the thesis opportunity and the nice people we had a chance to meet during our time at Axis.

Contents

1 Introduction **9**
- 1.1 Background . . . 9
- 1.2 Objectives and Research Questions . . . 10
- 1.3 Contributions . . . 10
- 1.4 Previous Work . . . 10

2 Approach **13**
- 2.1 Theory . . . 13
  - 2.1.1 Software Fault Prediction . . . 13
  - 2.1.2 Unsupervised Learning . . . 13
  - 2.1.3 Clustering Algorithms . . . 14
  - 2.1.4 Silver Standard Dataset . . . 16
  - 2.1.5 Curve Similarity . . . 16
  - 2.1.6 SZZ Algorithm . . . 17
  - 2.1.7 Performance Measures . . . 17
- 2.2 Method . . . 18
  - 2.2.1 Model Description . . . 18
  - 2.2.2 Analysis of Available Data . . . 18
  - 2.2.3 Choice of Metrics . . . 18
  - 2.2.4 Extraction of Features . . . 19
  - 2.2.5 Clustering . . . 20
  - 2.2.6 Validation Methods . . . 20
  - 2.2.7 Decision Boundaries . . . 21
  - 2.2.8 Predicting on New Data . . . 21

3 Dataset **23**
- 3.1 Features . . . 23
  - 3.1.1 Feature Importance . . . 24
- 3.2 Exploratory Data Analysis . . . 25
  - 3.2.1 Outlier Detection . . . 25

3.2.2 Normalization . . . 25
3.2.3 Correlation Analysis . . . 26
3.3 Dimensionality Reduction . . . 26
3.3.1 Principal Component Analysis . . . 28
3.3.2 t-SNE . . . 29

4 Results **31**
4.1 Cluster Validation on The *Technical Debt Dataset* . . . 31
4.1.1 K-Means . . . 32
4.1.2 DBSCAN . . . 32
4.1.3 Agglomerative Clustering . . . 33
4.1.4 Gaussian Mixture . . . 33
4.1.5 Fuzzy-C-Means . . . 33
4.1.6 Comparison of Clustering Algorithms . . . 35
4.2 Cluster Validation against Jira Issues . . . 35
4.3 Decision Boundaries . . . 37
4.4 Prediction Accuracy on New Data . . . 38
4.5 Programming Language Comparison . . . 39

5 Discussion **43**
5.1 Analysis of Results . . . 43
5.1.1 Choice of Clustering Algorithm . . . 43
5.1.2 Supervised Classifier Performance . . . 45
5.2 Cross-project Application . . . 47
5.3 Threats against Validity . . . 47
5.3.1 Programming Language Selection . . . 47
5.3.2 The SZZ Algorithm and The *Technical Debt Dataset* . . . 48
5.3.3 Inconsistency in Jira Issues . . . 48

6 Conclusions **51**
6.1 Future Work . . . 52

References **53**

Appendix A Decision Tree **59**

Glossary

change metrics Change metrics are metrics based on code changes, such as added lines of code and number of changed files.

clustering Clustering is an unsupervised machine learning technique to group objects in such a way that objects in the same group are more similar to each other.

fault In this report we define a fault as being a combination of bugs and failures/unexpected behaviour.

fault-proneness A fault-prone change is a change that is predicted to induce at least one fault in the system.

object oriented metrics Object oriented metrics are metrics based on class and design characteristics, such as class inheritance and coupling between objects.

software metrics Software metrics are metrics based on the code, such as number of for-loops or if statements in a class.

supervised machine learning Supervised machine learning use previously known input-output pairs to predict new outputs for different inputs.

unsupervised machine learning Unlike supervised machine learning, unsupervised machine learning has no prior knowledge of input-output pairs.

Chapter 1
Introduction

In this chapter, the background of our problem is explained and research questions are presented. We also list important previous work done in software fault prediction.

1.1 Background

When developing large scale software systems, extensive testing is a problem since the number of tests is large and a lot of time is allocated to these tests. To limit the amount of testing while still achieving a high test coverage, Software Quality Assurance (SQA) is a common industry practice (Grundy et al., 2015). SQA is a method for test engineers and developers to predict fault-prone areas with breakage or unexpected behaviour for every component of the system included in a new release. By doing this, more focus can be put on testing fault-prone components in an early state of development while other tests can be skipped. This process is called Test Case Prioritization (TCP). However, SQA is often based on developers and test engineers experience and human error is therefore a big factor. SQA is also very resource-heavy as it requires a lot of time. To minimize the risk of human error and resource usage, Software Fault Prediction (SFP) can be used as a guiding tool when assessing fault-proneness in different parts of the system.

Over the years, different models for fault prediction have been successfully developed and tried in software systems. The majority of these different models are based on supervised machine learning and three different kinds of metrics: software metrics, object oriented metrics and change metrics. Supervised learning requires that old fault data can be gathered and is sufficient enough to be used as labels in the model. Unsupervised learning on the other hand requires no previous knowledge and can be used on parts of the system which lacks sufficient fault data, for example on newer projects. Some work on unsupervised models using software metrics has been done (Catal et al., 2010), however no model using unsupervised learning in combination with change metrics has been explored regarding fault-proneness prediction. In this report, we present a novel SFP model based on change metrics and unsupervised learning.

1.2 Objectives and Research Questions

The aim of this thesis is to present a machine learning model predicting fault-proneness in a software system to assist developers with their test selection. This can be broken down in three research questions:

RQ1: Can an unsupervised machine learning model, predicting fault-proneness in a software system using change metrics, give comparable results to existing models presented in Section 1.4?

RQ2: If a model as the one presented in RQ1 exists, how can it be integrated with the system so that it functions cross-project and as the system evolves?

RQ3: Which clustering technique is optimal for fault-proneness prediction?

1.3 Contributions

Model Implementation

Both authors have worked actively with the implementation of the model. Elias was more focused on implementing the feature fetching from GitLab part of the script while Oskar worked with fetching features from Gerrit. Both authors worked evenly on the other parts of the model implementation.

Thesis Writing

Both authors have worked simultaneously on the thesis writing. Elias has in a large part contributed to the following sections: Abstract, 1.3, 1.4, 2.1.1, 2.1.2, 2.1.3, 2.1.5, 2.1.7, 2.2.1, 2.2.6, 3.1, 3.2.2, 3.2.3, 3.3, 4.1.1 - 4.1.5, 4.2, 4.3, 4.4, 4.5, 5.1.1, 5.1.2, 5.3.1 and Conclusions. Oskar has in a large part contributed to the following sections: 1.1, 1.2, 1.3, 1.4, 2.1.3, 2.1.4, 2.1.6, 2.1.7, 2.2.1, 2.2.2, 2.2.3, 2.2.4, 2.2.5, 2.2.6, 2.2.7, 2.2.8, 3.1, 3.1.1, 3.2.1, 4.1.1 - 4.1.5, 4.1.6, 5.1.1, 5.2, 5.3.2, 5.3.3, Conclusions and Future Work.

1.4 Previous Work

Software fault prediction has in recent years seen great development and promising results, both for supervised models and unsupervised models. In this section we present previous models that have influenced this thesis.

Abdeen et al. (2015) used the idea of *Change Impact Graphs* (CIGs), a type of object oriented metrics, to train a model to predict the impact a code change has on the software quality. They showed promising results using supervised learning with a precision average of 68% and a recall average of 46%.

Choudharya et al. (2018) built a fault prediction model solely based on change metrics i.e. the difference between two builds such as number of lines. They also analyzed existing and

new types of change metrics. Their results showed that change metrics give better prediction results than static code metrics in most cases.

The previous work presented this far only used supervised techniques as software fault predictors. However, Boucher and Badri (2018) presented different threshold models using unsupervised learning and software metrics to predict fault-proneness with high accuracy. They also presented the use of clustering techniques to group data and used threshold techniques to label the groups in terms of their fault-proneness.

Catal et al. (2010) presented clustering as an unsupervised method when little to no prior fault data can be used as labels in a supervised model. The authors used X-means clustering to decide the number of clusters (the value of k), that was used later when clustering with K-means. Their model used software metrics when training. They used False Positive Rate (FPR) and False Negative Rate (FNR) as performance metrics. The performance of their model gave a FPR of 34.55% and a FNR of 25%.

Liu et al. (2017) showed a model solving the Just-In-Time (JIT) prediction problem based on unsupervised learning and change metrics. JIT is used to predict fault-proneness of a change in the software system and does not give an estimate of the fault-proneness of single components. They showed an improvement using change metrics for JIT prediction over previously known models.

Caulo and Scanniello (2020) looked at a collection of works in the software fault prediction field and reviewed the different metrics they found. They also found which metrics that were no longer used and discussed the reasoning behind why certain metrics outlived others in the field.

Li et al. (2018) summarize typical work on software defect prediction. They discuss datasets (code metrics vs. process metrics), performance evaluation measures, prediction algorithms (supervised vs. semi-supervised vs. unsupervised), feature selection, data adoption, class imbalance and more.

Xu et al. (2021) listed and compared different research made on unsupervised software fault prediction models. They also compared unsupervised models to six different supervised models and found that some unsupervised models outperformed supervised models in some cases. In their report, they state that recent work in software fault prediction often use F-measure and AUC as performance metrics.

Yang et al. (2006) compared different clustering algorithms on different datasets. They showed that the choice of clustering algorithm is dependent on the dataset. They bring up the problem of using clustering on different datasets and concludes that "a proper clustering method will be effective to help software manager predict the software quality".

Nam and Kim (2015) created a complete software fault prediction model based on unsupervised learning. They came up with their own clustering strategy and used this to label the incoming data. The performance of their model gave an average F-measure of 0.636 and an average AUC score of 0.723.

Zhang et al. (2016) explored the performance for cross-project defect prediction by comparing unsupervised models with supervised model. They used software metrics as input for the models and AUC as performance measure when comparing models. The best model used spectral clustering and obtained a median AUC score of 0.71 across all projects.

The rest of the report is organized in the following way:

- Chapter 2 presents the theory and the methodology of the thesis.

- In Chapter 3 we discuss the dataset used is to train and validate the unsupervised model.
- In Chapter 4 we present our results in forms of plots and tables and in Chapter 5 the results are discussed.
- In Chapter 6 we present answers to the research questions.

Chapter 2
Approach

The aim of this chapter is to explain important theoretical concepts as well as the methods we used to answer the research questions.

2.1 Theory

This section gives the theoretical background of our methodology.

2.1.1 Software Fault Prediction

The reliability and quality of software are essential and depend on faults in the software. More faults will lead to a decrease in reliability and quality. Software quality assurance is an activity that is necessary to maintain high quality code. SQA often requires a great deal of resources and might be time consuming. Therefore, if SQA only has limited resources, it is an option to use software fault prediction before testing to improve SQA (Kumar and Rathore, 2018).

By utilizing SFP, fault-prone software modules can be predicted in a cost-efficient manner. The fault prediction model is typically trained on some structural or change-based properties and uses historical faults found in the project. With this trained model, it is possible to identify future fault-prone areas in the code in an early stage, which can save both time and resources.

2.1.2 Unsupervised Learning

There exist several different ways of performing the learning step in machine learning. One common technique is to perform unsupervised learning. It is called unsupervised because of the fact that there does not exist any labels (target values) for the data, unlike supervised

learning. In supervised learning, each input sample has a correct answer corresponding to that sample (Mishra, 2017).

The process of unsupervised learning is therefore to discover similarities between different samples in the data and make clusters or density estimations based on the features of the sample. The clusters can then be analysed and labeled by an expert if that is desired.

2.1.3 Clustering Algorithms

An important part of unsupervised learning is clustering, and there are many ways of producing the clusters. One thing to decide upon is the selection of clustering algorithm. While there exists many different clustering algorithms, none is a perfect fit for all types of problems and the problem at hand needs to be considered when choosing clustering algorithm. It is often required to know the number of clusters as it is an input to many clustering algorithms.

Here we describe the clustering algorithms that we decided to test and analyse. Other algorithms were also considered but they had hyperparameters that were difficult to tune in order to create a tool that works well on different projects with different datasets. We therefore omitted the clustering algorithms that we found to have very low potential.

K-Means

One popular and simple algorithm for clustering is ***K***-Means clustering. The algorithm takes one parameter as input defining the number of clusters, k. Here are the algorithmic steps (Sharma, 2019):

1. Randomly place the centroids.
2. Find the closest centroid to each sample and assign the sample to that cluster.
3. Calculate new cluster centroids based on the samples currently assigned to the cluster.
4. Repeat step 2 and 3 until convergence or a fixed number of iterations.

In order to determine the number of clusters that is optimal for the model, there are several methods that can be used. The elbow method is a heuristic algorithm that systematically tries different values of k. For each k, distances can be measured. The elbow of the curve is then used as a cutoff point. When dealing with clusters, this means that much better modeling of the data would not be achieved by adding more clusters.

DBSCAN

Another common clustering method is Density-Based Spatial Clustering of Applications with Noise (DBSCAN). The clustering works by finding samples that are densely grouped and putting them together. This makes DBSCAN useful when the ground truth clusters are of arbitrary shapes. According to Sharma (2020), it is therefore also really good at finding outliers in the dataset.

Unlike ***K***-Means, DBSCAN does not require to know how many clusters to identify. Instead, DBSCAN needs two other parameters, epsilon and min_samples. Epsilon specifies the radius for a circle that determines whether a sample is in the neighborhood of another

sample. The other parameter, min_samples, is the minimum number of required data points inside a circle to classify the sample as a core point.

Agglomerative Clustering

A third clustering method is the Agglomerative clustering. It is a form of hierarchical clustering which follows a bottom-up approach. The samples are at the beginning individual clusters and then pairs of clusters are merged together moving up the hierarchy, taking the closest cluster pairs first. This is repeated until one big cluster is formed, containing all samples.

This means that Agglomerative clustering can be used to explore a good value of the number of clusters. It is more simple to interpret compared to K-Means, since the constructed tree can be visualized for all different number of clusters and will have reproducible results (Pathak, 2018a). Representing the tree with a dendrogram will often make it easy to understand.

One problem with Agglomerative clustering is that it requires more storage and more computational power than the previously described clustering approaches. Another drawback with hierarchical clustering is that the algorithm can not undo any steps. Lastly the optimal number of clusters can be difficult to find if the dataset contains a huge amount of samples.

Gaussian Mixture with Expectation-Maximization

Gaussian Mixture Models (GMMs) can be seen as an extension to the K-Means clustering algorithm. While K-Means always creates spherical clusters based on distances, GMMs instead creates ellipse formed clusters based on Gaussian distributions. Since the clusters are based on Gaussian distributions, the final clustering is a soft clustering meaning that each point is assigned a probability to belong to each cluster.

As in K-Means it is necessary to define the number of clusters that is wanted. If k is the number of clusters, there are k Gaussian distributions. To find the mean, variance and density of these distributions the Expectation-maximization algorithm is used as explained by Sing (2019):

E-step: Calculate the probability that a point x_i belongs to a cluster $c_1, c_2, .., c_k$ with the following formula:

$$r_{ic} = \frac{\pi_c N(x_i; \mu_c, \sum_c)}{\sum_{c'} \pi_{c'} N(x_i; \mu_{c'}, \sum_{c'})},$$

where π_c is the density, μ_c is the mean and $\sum_c$ is the variance for a cluster c.

M-step: Update π, μ and $\sum$ according to the following formulas:

$$\begin{aligned} \pi &= \frac{\text{Number of points assigned to a cluster}}{\text{Total number of points}}, \\ \mu &= \frac{1}{\text{Number of points assigned to cluster}} \times \sum_i r_{ic} x_i, \\ \sum_c &= \frac{1}{\text{Number of points assigned to cluster}} \times \sum_i r_{ic} (x_i - \mu_c)^T (x_i - \mu_c). \end{aligned}$$

The E and M steps are repeated until the log-likelihood function is maximized.

Some advantages with using GMMs as clustering methods is the additional distribution information gained as well as the possibility to find clusters that are not spherical.

Fuzzy-C-Means

Fuzzy-C-Means (FCM) clustering can be seen as "soft K-Means" since the objective functions of the two algorithms are virtually identical. The difference between the two algorithms is that FCM calculates the inverse distance from a point to every cluster center to decide a membership value of that point to all clusters (Bezdek et al., 1984).

The membership value can be seen as a probability of a point belonging to a certain cluster since the distances are normalized between 0 and 1 and the algorithm is therefore a soft labeling algorithm.

The advantage of using Fuzzy-C-Means over K-Means is the ability to decide on a threshold of the membership value. What this means is that only points with a membership value for a certain cluster higher than a threshold will be put in that cluster. A disadvantage with using Fuzzy-C-Means over K-Means is that the algorithm is slower since it needs to calculate the inverse distance between every point and every cluster center.

2.1.4 Silver Standard Dataset

In software fault prediction, the lack of previously known fault data is a common problem and, in many cases, models are tested on different open source datasets. However, when using the model on a real project, training on an open source dataset with different properties than the dataset from the real project might cause problems.

Gathering accurate historical fault data from the real project is a time consuming process and counteracts the purpose of a software fault predictor. To work around the problem of gathering fault data, one can create a so called *silver standard dataset*, which is a dataset that has been automatically annotated. The annotation algorithm can be based on clustering so that the time constraint are minimal while still maintaining good understanding of the annotation algorithm.

The introduction of a silver standard dataset to any machine learning problem brings uncertainty to the validity of the created model. To solve this problem, an open source gold standard dataset can be used to validate the automatic annotation algorithm. A gold standard dataset is instead a dataset that is considered to be of the highest quality possible.

2.1.5 Curve Similarity

There exist many measurements that calculate a similarity between two curves. The two measurements used in this report are:

Partial Curve Mapping (PCM) uses both area and arc-length in combination when determining the similarity between two curves (Jekel et al., 2019).

Dynamic Time Warping (DTW). To get an even better measurement of the similarity, the time can be removed from the equation. That is because the Jira issue curve should be time shifted (appear after the fault-inducing commits) and not occur at the same time. This is solved by using DTW that measures similarity between two sequences that may vary in speed or time (Kyaagba, 2018).

2.1.6 SZZ Algorithm

The SZZ Algorithm developed by Śliwerski et al. (2005) is a widely used algorithm in fault prediction problems. The algorithm is often used in supervised learning methods when assigning labels since its main function is to identify fault-inducing commits from a projects version history. The algorithm consists of three steps. Firstly, all fault-fixing commits are identified by scanning commit messages. The second step of the algorithm looks at the diff of the fault-fixing commit to decide what fixed the fault and how the data-structures of the project have changed. In the last step of the algorithm, the commits that caused the fault are determined by using the annotate/blame tool in Git.

2.1.7 Performance Measures

In this section, we present the different performance measures used to evaluate our model.

Precision, Recall and F-score

To decide which clustering algorithm to choose, we used precision and recall as evaluation metrics for different clustering algorithms. We wanted a clustering algorithm with high recall while still maintaining relative high precision. A high precision often comes at the cost of a low recall and vice versa. Precision is in our experiment the proportion of predicted fault-prone commits that actually were fault-inducing commits out of all commits we predicted as fault-prone. Recall is the proportion of predicted fault-prone commits that actually were fault-inducing commits out of all actual fault-inducing commits.

		Predicted	
		Safe	Fault-Prone
Actual	Safe	TN	FP
	Fault-Inducing	FN	TP

- TN = True Negative. Predicted safe, actual safe.
- FP = False Positive. Predicted fault-prone, actual safe.
- FN = False Negative. Predicted safe, actual fault-inducing.
- TP = True Positive. Predicted fault-prone, actual fault-inducing.

$$Precision = \frac{TP}{TP + FP} = \frac{\text{Predicted fault-prone, actual fault-inducing}}{\text{Predicted fault-prone}}$$

$$Recall = \frac{TP}{TP + FN} = \frac{\text{Predicted fault-prone, actual fault-inducing}}{\text{Actual fault-inducing}}$$

It is common to measure the accuracy of a test using the F-score. If precision and recall are of equal importance, the F_1 measure can be used. It is the harmonic mean of precision and recall. But if they are not of equal importance, the equation can be modified. The F_2

score is for example defined to consider recall twice as important as precision. In general, the F_β score can be written as follows.

$$F_\beta = (1+\beta^2) \cdot \frac{precision \cdot recall}{(\beta^2 \cdot precision) + recall} = \frac{(1+\beta^2) \cdot TP}{(1+\beta^2) \cdot TP + \beta^2 \cdot FN + FP}$$

Area Under Curve

In software fault prediction, it is common practise to use the Area Under the Receiver Operator Characteristics Curve (AUC) as a performance metric. The Receiver Operator Characteristics Curve (ROC Curve) is a curve that show the trade-off between false positive rate and true positive rate for different decision thresholds. The characteristics of the ROC curve makes it a good performance metric when a severe class imbalance is present. It means that the dataset does not contain equally many samples for the two classes, a common occurrence with SFP problems (Fawcett, 2006).

AUC is the area under the ROC curve and is used to get a single comparable value to measure the diagnostic ability of a binary classifier.

2.2 Method

In this section, we explain in detail our methodology and experimental setup.

2.2.1 Model Description

The first step of our model was to choose a fitting dataset that would benefit from an unsupervised model. Secondly the model had to label the data either fault-prone or safe with clustering, creating a silver standard labeled dataset. To do this, we chose a set of features described in Table 3.1 and mined them from the chosen projects. The last step of our model was to use the silver standard data to train a supervised classifier to predict new data points as either fault-prone or safe, see Figure 2.1.

2.2.2 Analysis of Available Data

To get an understanding of what types of machine learning algorithms we could use to solve our problem, an extensive analysis of the available data had to be done. Since Axis is a large company with many different products, we decided to build our model based on data from one of Axis' products. The product in question was fairly new and a lot of prior fault data was not present. This indicated that a model based on supervised learning would have had a small dataset to train on and a model based on unsupervised learning and clustering was more suitable.

2.2.3 Choice of Metrics

To be able to make a model work cross-project, the complexity of feature extraction had to be taken in to consideration when choosing what type of features were suitable for our model.

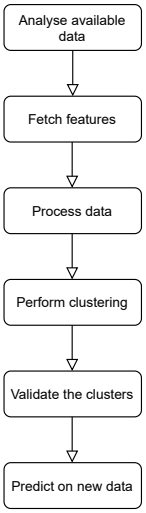


Figure 2.1: A brief overview of our model.

The product was built from mainly three different components and the code was spread out over different repositories in GitLab and Gerrit. Because of the many different parts and repositories in the system, we ruled out object-oriented metrics based on dependencies since the complexity and resources needed to find dependencies at a release would outweigh the resources needed for standard SQA.

The software of the three different components were written in many different programming languages and since software metrics are highly language dependent, we ruled out these types of metrics as well.

The APIs for both GitLab and Gerrit have great support for extraction of change metrics at every commit of new code. This makes the process of feature extraction and retraining of the unsupervised model cheap in terms of resources. While change metrics still are language dependent, the dependency is not as strong as for software metrics. Therefore, we chose change metrics as features to the model.

2.2.4 Extraction of Features

We extracted the features used in the model through the GitLab and Gerrit APIs. Both APIs are query based and Table 2.1 lists the queries sent to extract different features.

Query	Explanation
/changes/?q=status:merged+project:package_path	Get all merged commits from a specified repository in Gerrit
/projects/repository_id/repository/commits?	Get all merged commits from a specified repository in GitLab
/changes/'changeid'	Get all info about a specified commit in Gerrit
/projects/repository_id/repository/commits/commit_id/diff	Get files modified, lines added and lines deleted for a specified commit in GitLab
/changes/?q=file:file_path	Get the history of a specified file in Gerrit
/projects/repository_id/repository/commits?path=file_path	Get the history of a specified file in GitLab

Table 2.1: Queries used to extract features.

Combining these queries, we extracted the features listed in Table 3.1 for all commits made to the different components' respective software. The feature selection was based on two things: previous work and accessibility. When studying what features other people extracted in similar research, we decided to choose those that seemed to be the most important

and gave the best results. Of course, we also had the limitation of features based on what was accessible at Axis.

We excluded code not written in C, TypeScript, JavaScript or Go since those languages were used in the product and we suspected that different languages gave different feature values. If data from more than four languages were used, the created clusters would have been hard to interpret.

The final dataset was then saved to a CSV file and is described in detail in Chapter 3.

2.2.5 Clustering

When we had completed our dataset, the next step was to try the different clustering algorithms presented in Section 2.1.3 and analyse the clusters to decide which clustering algorithm to use. We clustered commits to create a silver standard dataset, labels from this dataset were later used in the predicting part of our model.

We first normalized the data and removed outliers as explained in Section 3.2.1. We then trained a different model for every clustering algorithm on all features, see Table 3.1.

Code for the different clustering algorithms was imported from scikit-learn (Pedregosa et al., 2011). Code for the Fuzzy-C-Means algorithm was written by Dias (2019).

2.2.6 Validation Methods

In this section, we explain the two different methods used to validate the clusters.

Validation against Jira

Jira is a tool that can be used for issue tracking in software development projects. When a fault in the system is found, it is reported and documented. In our case, this meant that all found faults in the system were listed with the date and time the issue was created, and also in which specific component the fault was likely to be.

With this information about the issues, we were able to compare known issues with the commits we labeled as fault-prone. We created curves representing commits in a specific cluster and compared it to a curve representing the number of issues in Jira, using different curve similarity metrics.

Validation against The *Technical Debt Dataset*

The *Technical Debt Dataset* is a dataset mined by Lenarduzzi et al. (2019) consisting of different measures on all commits of 33 Apache Software Foundation projects in Java.

Lenarduzzi et al. (2019) used SZZ to label every commit as either fault-inducing or not fault-inducing as well as Git version history to extract change metrics.

Since all the projects in the *Technical Debt Dataset* were written in Java and differed from the project we used to train our model, we could not use the dataset to directly validate our trained model. Instead we trained a new model for the collection of projects with the same features used in our original model to validate our methodology and our feature selection.

By looking at the distribution of all commits and comparing to the distribution of fault-inducing commits over the different clusters, we were able to see if the fault-inducing commits were part of the cluster we marked as fault-prone. We performed this procedure for all libraries in the dataset to get an overall picture of how well our features worked as predictors.

2.2.7 Decision Boundaries

Since our original model was based on unsupervised learning and clustering, we came up with a hypothesis on what the different clusters represented.

To strengthen this hypothesis, we used the *Technical Debt Dataset* and supervised learning to build a decision tree for every unique feature. We did this by training a model with one single feature at a time.

With this method, we found soft decision boundaries for every feature; soft meaning that we only cared if a higher or lower value of the feature gave a fault or not. However, since we wanted to get reliable boundaries, we had to retrain a decision tree classifier for every feature until the accuracy and recall were high enough for the boundary to be considered reliable.

When we had found the different decision boundaries, we compared them with the position of our different cluster centroids. By doing this, we were able to see if points in a cluster laid over or under the boundary for fault-inducing commits and could therefore strengthen our hypothesis of what the different clusters represented.

2.2.8 Predicting on New Data

In machine learning, supervised learning is the task of learning a function that maps an input to an output based on previously known input-output pairs. For SFP problems, supervised machine learning algorithms are often referred to as binary classifiers since they classify an input as either safe or fault-prone.

The last step of our model in Figure 2.1 “Predict on new data” was performed by training a binary classifier on the training data produced in the clustering step on the Axis dataset. This dataset is after clustering considered a silver standard dataset. We evaluated the performance of four different classifiers, $\boldsymbol{K}$-Nearest Neighbors, XGBoost, Random Forest from scikit-learn (Pedregosa et al., 2011) and LightGBM by Zhang et al. (2017).

Chapter 3
Dataset

We built the dataset, used to train our model, with every commit ever made to the software of the Axis product. From these commits, we extracted the features in Table 3.1.

3.1 Features

Feature	Explanation
LOC-ADDED	The total number of lines of code added in the commit
LOC-DELETED	The total number of lines of code deleted in the commit
MAX-LOC-WORKED-ON	The maximum number of lines worked on (additions + deletions) among the files in the commit
AVG-LOC-WORKED-ON	The average number of lines worked on (additions + deletions) between the files in the commit
TOTAL-CHANGED-FILES	The total number of changed files in the commit
MAX-COMMITS-TO-FILE	The maximum number of previous commits a file in the commit has been changed in
AVG-COMMITS-TO-FILE	The average number of previous commits the files have been changed in
MAX-AUTHORS	The maximum number of authors that have made changes to one of the files in the commit
AVG-AUTHORS	The average number of authors that have made changes to the files in the commit
COMMIT-COUNT-BY-AUTHOR	The total number of commits the author has contributed with in the project

Table 3.1: Extracted features.

The initial features that we decided to extract for each commit are shown in Table 3.1. After analysing every feature individually, we decided to remove one of the features, COMMIT-COUNT-BY-AUTHOR. Our hypothesis was that all features were more fault-prone the higher value they had, the only exception being COMMIT-COUNT-BY-AUTHOR. Training a simple decision tree classifier on the *Technical Debt Dataset* can show how decision boundaries are created for all features. This reveals whether our hypothesis is correct or not.

Figure 3.1 shows that COMMIT-COUNT-BY-AUTHOR did not match the hypothesis that we had. Our belief was that a higher value would imply that the developer creating the commit had more experience (a large amount of earlier commits) and a low value would indicate more fault-prone commits. But instead, large values seem to mean more fault-prone

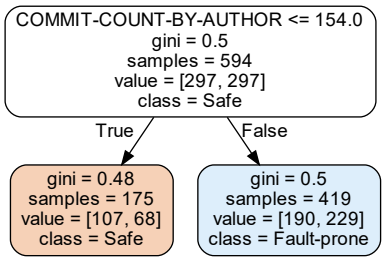


Figure 3.1: The decision tree for COMMIT-COUNT-BY-AUTHOR feature.

commits. This was not the expected behaviour therefore we removed it. All other nine features seemed to agree with what we expected, so the dataset ended up with the first nine features. The decision boundaries for the nine chosen features will be described in more detail in Chapter 4.

3.1.1 Feature Importance

To get an understanding of the importance of each individual feature, we looked at the feature importance after we had trained a supervised classifier on the labeled (clustered) data. We did this for three different classifiers: LightGBM, XGBoost and Random Forest, using Fuzzy-C-Means as the clustering algorithm.

In Table 3.2 the individual importance of each feature for the three different algorithms is presented. The results indicates that features measuring the size of a change were of greater importance than features connected to file history.

Feature	LGBM	XGBoost	Random Forest
LOC-ADDED	**0.17972**	**0.22988**	0.15559
LOC-DELETED	0.15024	0.15104	0.07199
MAX-LOC-WORKED-ON	0.16001	0.14523	**0.20662**
AVG-LOC-WORKED-ON	0.12751	0.12365	0.09380
TOTAL-CHANGED-FILES	0.08080	0.05560	0.05536
MAX-COMMITS-TO-FILE	0.10389	0.10373	0.14814
AVG-COMMITS-TO-FILE	0.09679	0.09046	0.06778
MAX-AUTHORS	0.05452	0.04315	0.13721
AVG-AUTHORS	0.04653	0.05727	0.06350

Table 3.2: Importance for each individual feature using different supervised classifiers.

3.2 Exploratory Data Analysis

Exploratory Data Analysis (EDA) is the name of the process of understanding and visualizing previously unknown datasets. In machine learning, EDA is the process of finding outliers in the dataset, correlation between different features and skewness of features. In this section, we present our results after performing EDA on the Axis dataset.

3.2.1 Outlier Detection

Outlier detection is the process of identifying rare points in the dataset that differs greatly from all other points.

An example of an outlier with the chosen features dataset could for example be a huge refactoring of names, where a lot of files were touched or a commit with a test file that was included as standard in some commits and therefore got big MAX-AUTHORS and MAX-COMMITS-TO-FILE values. Keeping these outliers would greatly affect our results and therefore we performed outlier removal.

To find outliers in our data, we initially placed all the points in a data frame, where every row was a unique data point (commit) and all of the columns represented the values of the different features. We calculated Z-scores, the number of standard deviations away from the mean, for every point and the points with at least one feature with a Z-score lower than -3 or higher than 3 were removed. Figure 3.2 shows the different TOTAL-CHANGED-FILES values before outlier removal and Figure 3.3 shows values for the same feature after outlier removal. After outlier removal, the dataset was reduced from 3188 to 3029 data points.

Figure 3.2: A box plot of the TOTAL-CHANGED-FILES points before removal of outliers.

3.2.2 Normalization

Normalization of the data is a very common technique when it comes to data preprocessing. There are two frequently used methods, Z-normalization (often called standardization) and Min-Max normalization. Z-normalization transforms the data to have zero-mean and unit variance while Min-Max normalization rescales the data into the range [0, 1]. If the data for different features differ in magnitude, the features with large values will be treated as more

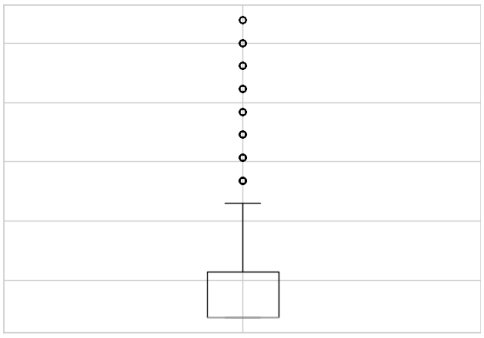


Figure 3.3: A box plot of the TOTAL-CHANGED-FILES points after removal of outliers.

important than features with smaller values, when it comes to distance-based algorithms (Zhang, 2019).

Many clustering algorithms are based on Euclidean distance. The optimal normalization method for such a problem is, according to Zhang (2019), standardization if it is desired to have equal feature importance. Therefore we decided to standardize feature data after outlier removal.

3.2.3 Correlation Analysis

A correlation analysis shows the correlation between features and can be useful to decide if some features can be discarded as they are to strongly correlated with other features.

Figure 3.4 shows the correlation matrix of our features. Some of the features are strongly correlated, however these features were expected to be strongly correlated when they were chosen as features. For example from Figure 3.4, it can be seen that MAX-LOC-WORKED-ON and AVG-LOC-WORKED-ON are strongly correlated and this is to be expected since a commit with only one file will have the same value for both MAX-LOC-WORKED-ON and AVG-LOC-WORKED-ON. This was the case for most of the feature pairs that were strongly correlated. Since these correlations were expected, we decided to keep all of the features.

In Figure 3.4 it is seen that we also included pc1 and pc2 in the correlation analysis. Those are not features, but they represent the two first principal components. The first principal component shows large correlation with the last four features, while the second principal component shows high correlation with the five first features. This will be discussed in more detail in Section 3.3.1.

3.3 Dimensionality Reduction

In machine learning, it is very common for datasets to contain a large number of features. When the number of features increases, so does the dimensionality. Having too many features (high dimensionality) compared to the number of samples introduces an increased risk of overfitting the model. This is called the curse of dimensionality (Yiu, 2019). Overfitting means that the model is trained to fit the training data very well but does not perform well on new data.

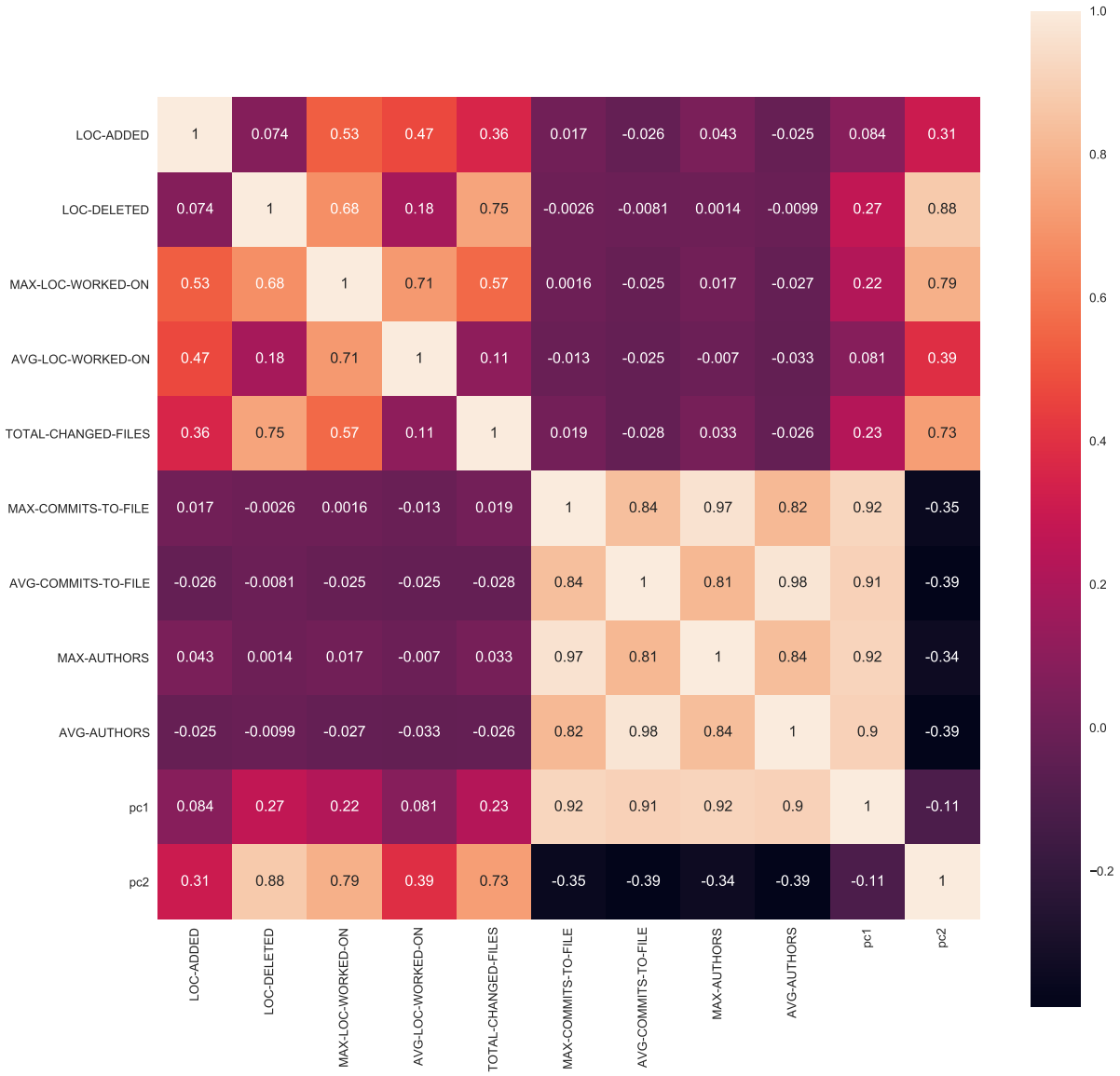


Figure 3.4: Correlation matrix of our features.

Reducing the number of features will give fewer variable relationships to train and the model will be less prone to overfit. One method to reduce the number of features is to simply perform a feature selection and excluding some of them. Dimensionality reduction is instead a transformation of all the existing features into a lower dimension.

In our case, we decided to have only nine features, which is not a lot, and therefore it is not necessary to perform dimensionality reduction to reduce overfitting. But in order to visualize the data we would need to do it in nine dimensions. That is not simple to do or interpret. Instead we only use dimensionality reduction techniques to reduce the number of dimensions to two so the resulting clusters can be visualized and interpreted more easily in 2D. Here, two techniques to reduce the dimensionality are described:

1. Principal component analysis and

2. t-distributed stochastic neighbor embedding.

3.3.1 Principal Component Analysis

The goal of Principal Component Analysis (PCA) is to reduce the feature space dimension into a number of principal components that are linear and orthogonal towards each other. PCA is trying to put maximum information in the first principal component, then the remaining maximum information in the second principal component and so on. This is equivalent to finding the direction of a line that best fits the data while still being perpendicular to the previous lines (principal components). It will therefore minimize the average squared distances between the points and the line.

Using the calculated principal components as a new basis for the data can be very useful when visualizing the data. If only two or three principal components are used then the data can be plotted in 2D and 3D respectively. Of course some information is lost during PCA, but since the information is prioritized to be described by the first principal components, it is possible to keep a majority of the explained variance with only a few principal components. Therefore, a good representation of the data can often be visualized in only two dimensions (Brems, 2017).

Performing PCA will also come with the cost that the basis will be less interpretable. After PCA, the principal components (often visualized as axis in a coordinate system) will no longer represent a certain feature. It will instead be a combination of several features.

In Figure 3.5, the individual explained variance for every principal component for our dataset is visualized. From the figure, it can be seen that roughly 70% explained variance is kept when only the two first principal components are used.

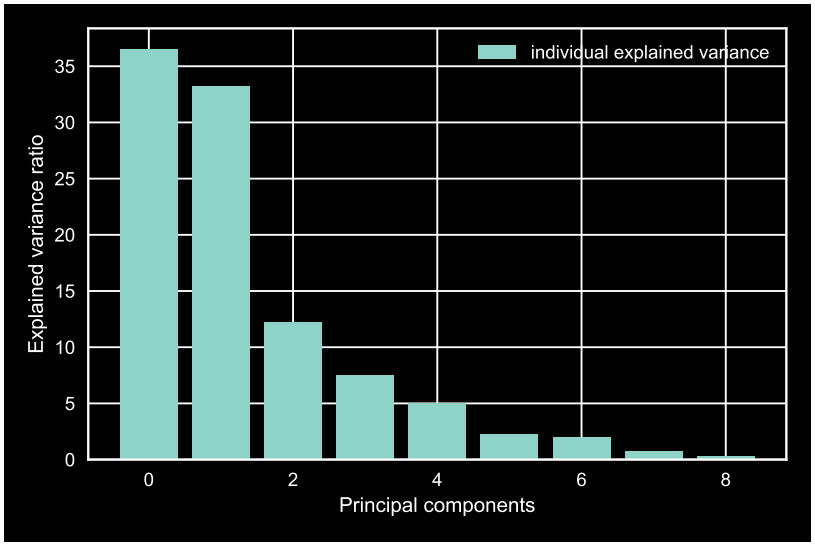


Figure 3.5: The individual explained variance for our principal components.

3.3.2 t-SNE

Just like PCA, t-distributed Stochastic Neighbor Embedding (t-SNE) is used for reducing the number of dimensions of a dataset. But how it is done is a bit different. Here are some differences between t-SNE and PCA (Pathak, 2018b):

- t-SNE is based on conditional distributions (probability theory). The divergence between two distributions are being minimized. PCA, on the other hand, is a mathematical model based on linear algebra.
- The computational cost is significantly higher for t-SNE than for PCA. So when dealing with large datasets PCA will be much faster than t-SNE.
- t-SNE is non-linear while PCA is linear. It means that t-SNE could find polynomial relationships between features that PCA can not find.
- t-SNE has parameters that needs to be tuned in order for the dimensionality reduction to work properly, for example learning rate and perplexity. PCA only needs the number of principal components to create.
- Unlike PCA, t-SNE does not necessarily produce the same result, running the exact same program multiple times.

In our case, the first two principal components after performing PCA will actually have large significance, while the dimensionality reduction from t-SNE does not. In addition to this, t-SNE is slower and have more parameters to tune. Those arguments made it easy to choose PCA as dimensionality reduction technique for visualization of scatter plots in the rest of this report.

Chapter 4

Results

4.1 Cluster Validation on The *Technical Debt Dataset*

In this section, we present the different clusters created for different clustering algorithms. To get an initial understanding of the number of clusters the data should be divided into, we used the elbow method for K-Means, see Figure 4.1. When results for other algorithms than K-Means were analysed, we found that more than two clusters were redundant or gave worse results.

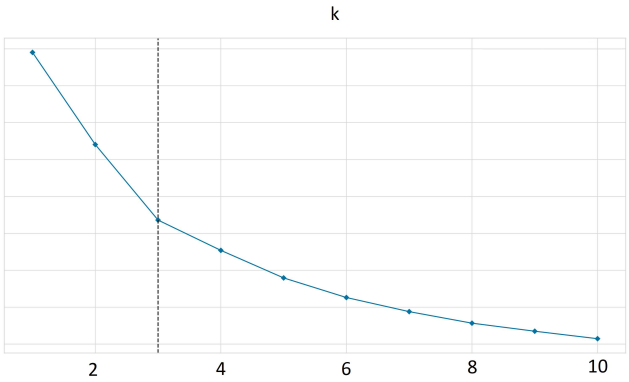


Figure 4.1: Result from elbow analysis.

In the following results from different clustering algorithm, we used the same Axis dataset, containing 3029 commits.

4.1.1 K-Means

Figure 4.2 shows the clusters created with *K*-Means algorithm with the number of clusters k set to 2. Here, 15.6% of the commits were classified as part of the red cluster (top) and 84.4% of the commits were classified as part of the green cluster (bottom).

Figure 4.3 shows the clusters created with *K*-Means algorithm when k, the number of clusters, where set to 3. Here, 13.9% of the samples are a part of the blue (bottom right) cluster, 13.7% are a part of the red (top left) cluster and 72.4% are a part of the green (bottom left) cluster. Here both the red and blue clusters are considered fault-prone.

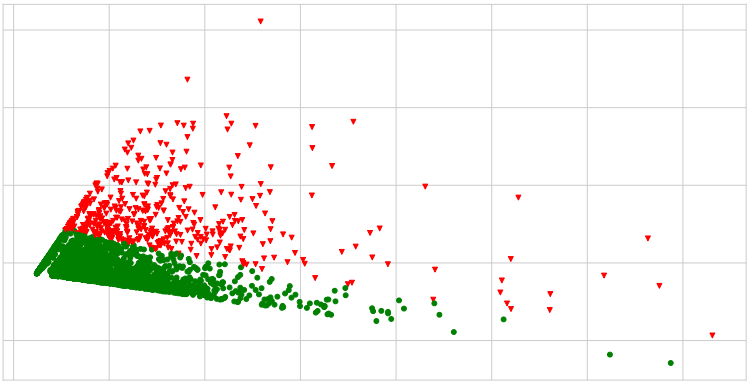


Figure 4.2: Created clusters with *K*-Means algorithm plotted over the two principal components.

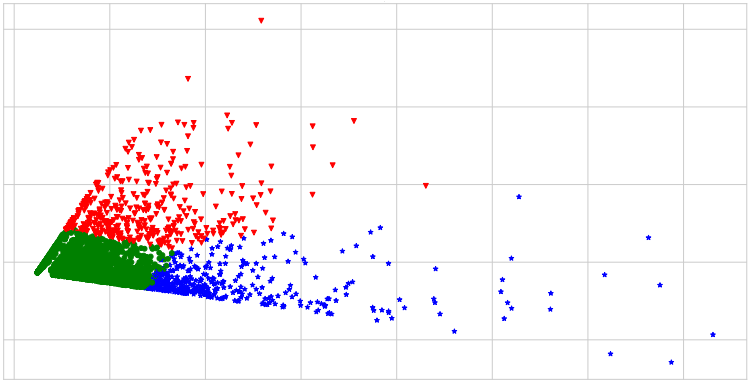


Figure 4.3: Created clusters with *K*-Means algorithm plotted over the two principal components.

4.1.2 DBSCAN

Figure 4.4 illustrates a clustering created with the DBSCAN algorithm. With default parameters, 69.7% of all commits are in the safe green (bottom left) cluster, and 30.3% in the fault-prone red (upper right) cluster.

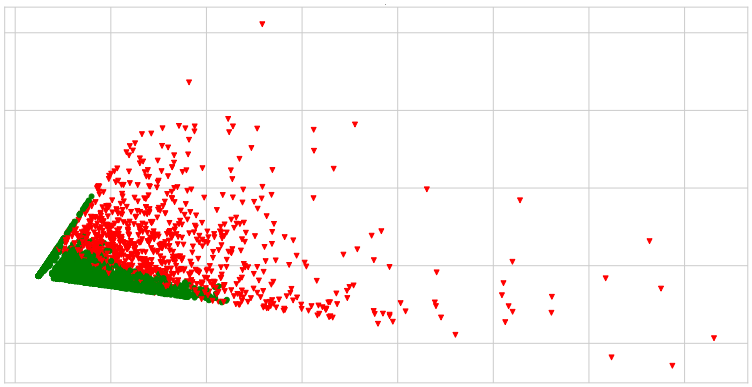


Figure 4.4: Created clusters with DBSCAN algorithm plotted over the two principal components.

4.1.3 Agglomerative Clustering

Figure 4.5 visualizes a clustering created with the Agglomerative clustering algorithm. The number of clusters parameter was set to 2. Here, 77.1% of all commits are in the safe green (bottom) cluster, and 22.9% in the fault-prone red (top) cluster.

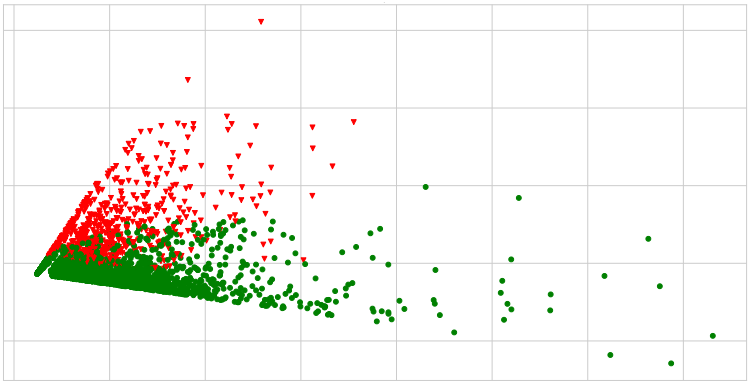


Figure 4.5: Created clusters with Agglomerative clustering algorithm plotted over the two principal components.

4.1.4 Gaussian Mixture

Figure 4.6 shows the clusters created with Gaussian mixture. Here, 58.1% of the commits are a part of the safe green cluster (bottom left) while 41.9% of all commits are a part of the fault-prone red cluster (upper right).

4.1.5 Fuzzy-C-Means

Figure 4.7 shows the clusters created with Fuzzy-C-Means clustering: 58.3% of the commits are located in the green cluster (bottom left), and 41.7% are considered fault-prone as they are in the red (upper right) cluster. Figure 4.8 shows the Receiver Operating Characteristic (ROC) curve. For this ROC curve, the AUC score was 0.6800.

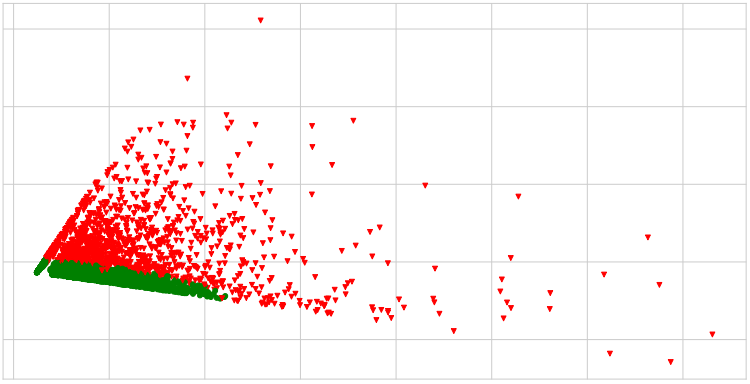


Figure 4.6: Created clusters with Gaussian mixture algorithm plotted over the two principal components.

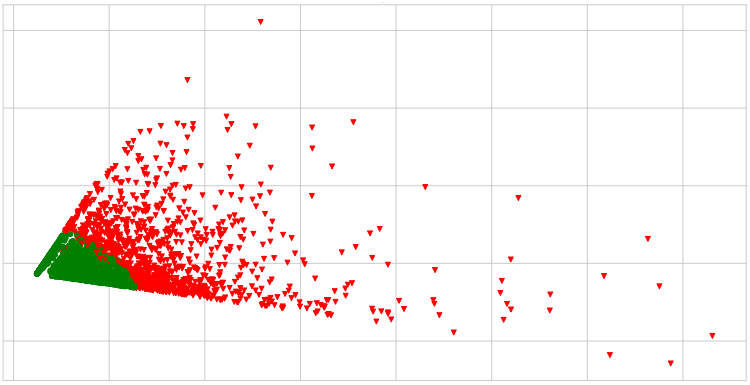


Figure 4.7: Created clusters with Fuzzy-C-Means Clustering algorithm plotted over the two principal components.

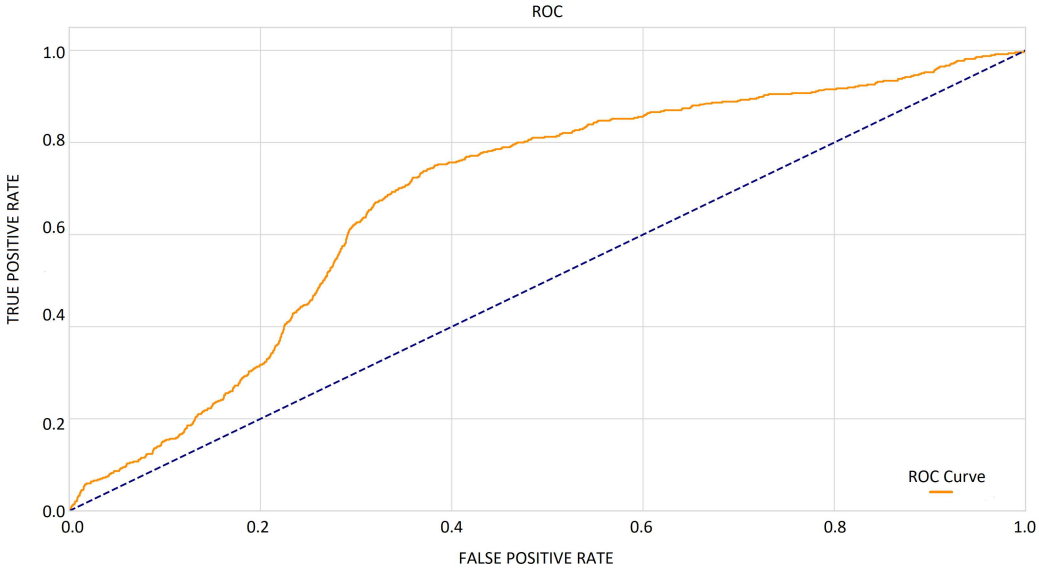


Figure 4.8: The ROC curve of the Fuzzy-C-Means clustering.

4.1.6 Comparison of Clustering Algorithms

Tables 4.1 and 4.2 present recall and precision for different clustering algorithms and with different distributions of safe and fault-inducing commits.[1] With the *Technical Debt Dataset*, recall and precision could be measured and a comparison between all models was made.

By the look of Figures 4.2-4.7, a good clustering is assumed to follow the hypothesis that larger feature values would imply more fault-prone commits. With this in mind, DBSCAN and Fuzzy-C-Means seems to cluster the data better compared to the other models.

In the *Technical Debt Dataset* the proportion of fault-inducing commits is slightly above 1%. Therefore we assume that, in large software systems, it is more plausible that the distribution is 90/10 or 99/1 rather than 50/50. We also found recall to be slightly more important than precision since it is of higher interest to find fault-prone areas of the system and not to identify a very small proportion of faults with higher precision.

With this in mind, Fuzzy-C-Means was found to be the best algorithm for our model in those cases and therefore all future analysis was done with Fuzzy-C-Means clustering. The choice of Fuzzy-C-Means is discussed in more detail in Section 5.1.1.

Method (Number of Clusters)	Precision 50/50	Precision 60/40	Precision 70/30	Precision 80/20	Precision 90/10	Precision 99/1
K-Means (2)	0.8267	**0.8095**	0.7011	**0.6212**	**0.4057**	0.0226
K-Means (3)	0.7508	0.6412	0.5324	0.3961	0.2073	0.0215
DBSCAN (2)	0.6377	0.5642	0.4511	0.3573	0.2088	**0.0318**
Agglomerative (2)	**0.9167**	0.7843	**0.7014**	0.5041	0.3647	0.0138
Gaussian Mixture (2)	0.6969	0.6244	0.4788	0.3350	0.1662	0.0173
Fuzzy-C-Means (2)	0.7465	0.6336	0.5421	0.3588	0.1841	0.0192

Table 4.1: The precision of fault-prone class, with different safe/fault-inducing distributions, for different clustering algorithms.

Method (Number of Clusters)	Recall 50/50	Recall 60/40	Recall 70/30	Recall 80/20	Recall 90/10	Recall 99/1
K-Means (2)	0.1278	0.1402	0.1258	0.1691	0.1773	0.5670
K-Means (3)	0.5031	0.5196	0.5423	0.5505	0.5979	0.6433
DBSCAN (2)	**0.8021**	**0.7794**	**0.7711**	**0.7588**	0.7072	0.5588
Agglomerative (2)	0.0680	0.0825	0.2082	0.2515	0.1278	0.4495
Gaussian Mixture (2)	0.5546	0.5588	0.6041	0.6990	0.7216	0.7361
Fuzzy-C-Means (2)	0.5526	0.6062	0.6371	0.6887	**0.7237**	**0.7464**

Table 4.2: The recall of the fault-prone class, with different safe/fault-inducing distributions, for different clustering algorithms.

4.2 Cluster Validation against Jira Issues

Figures 4.9, 4.10 and 4.11 show the number of cluster points and the number of issues in Jira over a time frame for each of the components. The green line represents the number of commits in the safe cluster (y-axis) at a specific time point (x-axis). The red line is instead

[1] In the tables 90/10 for example, means that 90% of the commits in the dataset are safe and 10% are fault-inducing.

representing commits in the fault-prone cluster. The black line represents the number of Jira issues at a specific time point. To compare the lines, they were all normalized to have an area under curve equal to one.

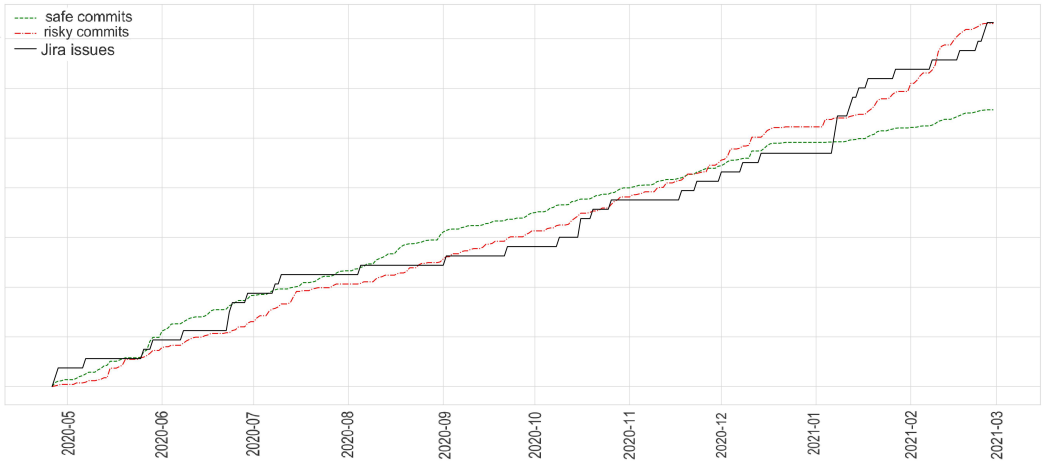


Figure 4.9: A plot of the commit count and Jira issue count for component 1.

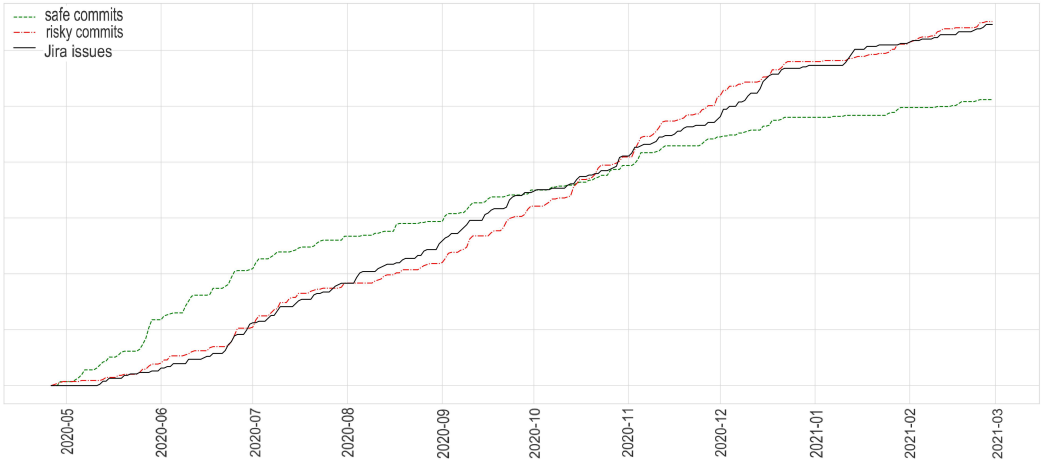


Figure 4.10: A plot of the commit count and Jira issue count for component 2.

We calculated two different similarity measures, dynamic time warping and partial curve mapping, for all components using the plotted lines. Table 4.3 shows the dynamic time warp distance. For all components, the distance is lower between the red and black line compared to the distance between the green and black line. This means that the red line is more similar to the black line than the green line is. In Table 4.4, the partial curve mapping between the Jira curve and a cluster lines are presented. Also here the differences show that the red line is the most similar to the black line.

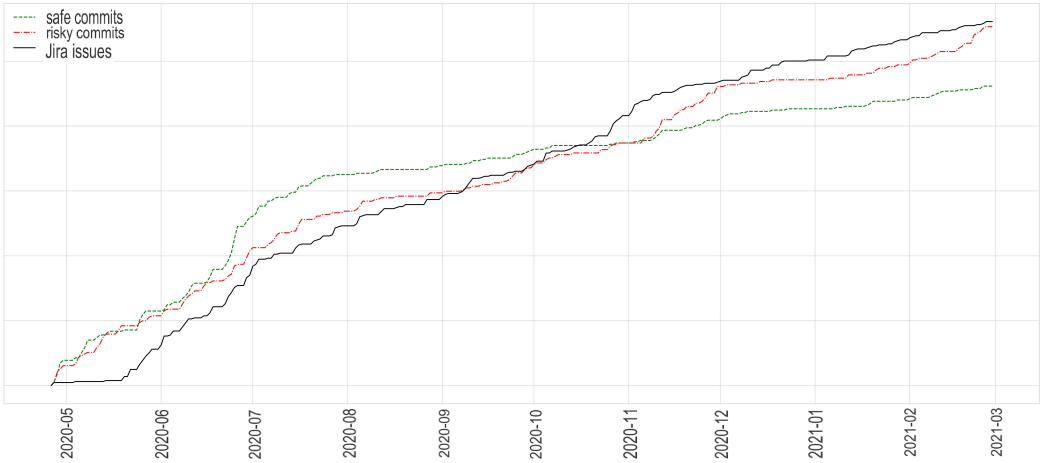


Figure 4.11: A plot of the commit count and Jira issue count for component 3.

	Component 1	Component 2	Component 3
Green line	0.136	0.198	0.193
Red line	0.084	0.053	0.070

Table 4.3: The Dynamic Time Warp distance between each cluster and the number of issues in Jira.

	Component 1	Component 2	Component 3
Green line	11.38	23.95	23.56
Red line	4.09	3.51	4.68

Table 4.4: Partial curve mapping for Jira and cluster curves.

4.3 Decision Boundaries

To validate our hypothesis that each of our nine features are more fault-prone for higher values, we determined to test all of them, separately, using a decision tree classifier. The *Technical Debt Dataset* was randomly undersampled so that the two classes, safe and fault-prone, were of equal size. Then we trained the classifier with the max depth parameter set to one for all individual features. The result is shown in Figure 4.12. Here it is clear that a higher value of each feature result in a more fault-prone commit.

Take LOC-ADDED as an example. The white box shows that the tree in total contains 572 samples, equally distributed between the classes. The decision threshold is whether the value of LOC-ADDED is smaller or equal to 31.5. If it is true, then the sample will go to the left box, otherwise to the right. The left box ended up with 205 samples with 152 samples of them actually being safe, and the rest 53 samples were fault-prone. There are nearly three times more safe commits than fault-prone and therefore the left box is classified as safe (the decision is majority based). The distribution in the right box was 134 safe and 233 fault-prone. That makes the samples in the right box being classified as fault-prone.

An example of a full decision tree, using all features is visualized in Figure A.1. Here it

can be seen that a value above the threshold does not always give more fault-prone commits and vice versa, but in general it is clear that the left side of the tree is more reddish (classified as safe) and the right side is more bluish (classified as fault-prone).

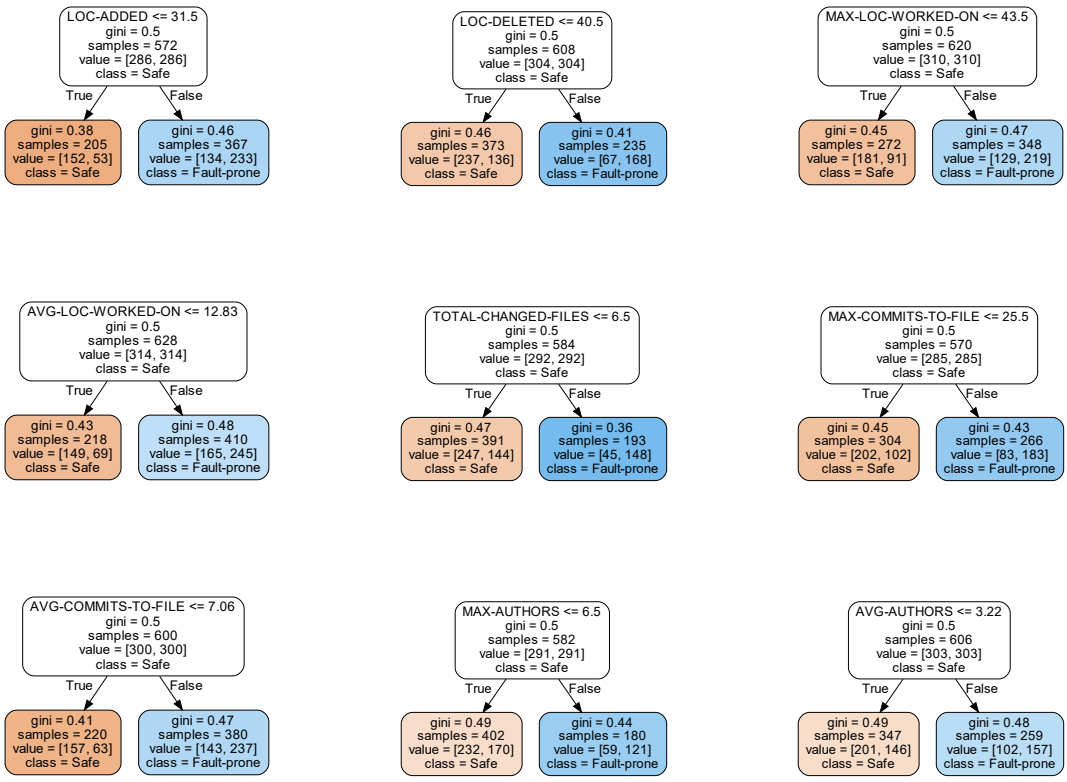


Figure 4.12: Decisions for each feature.

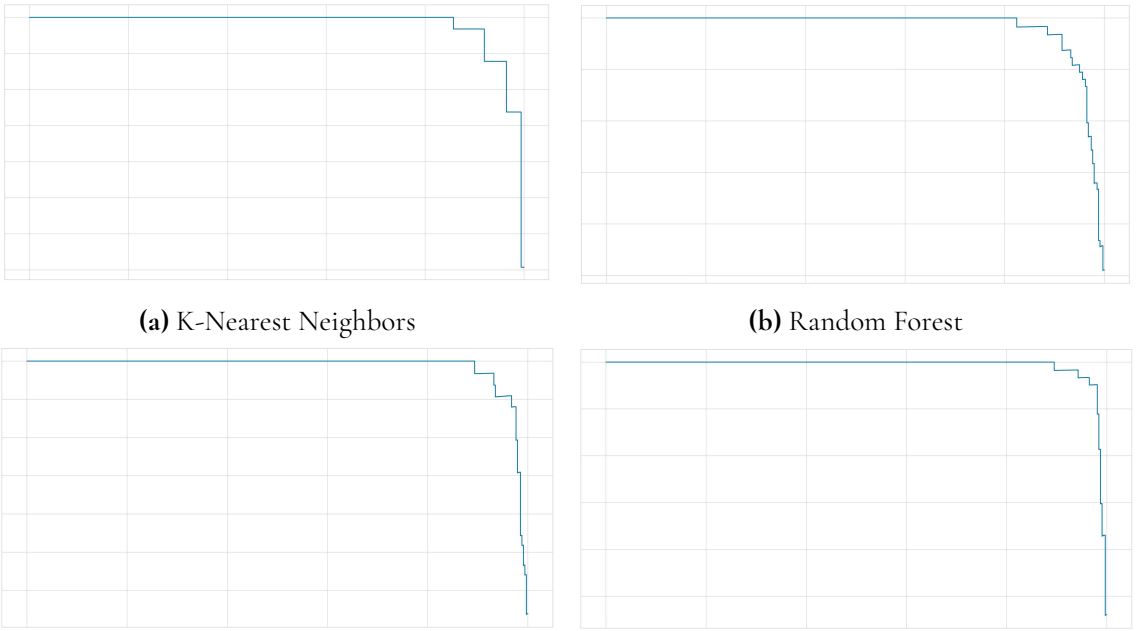
4.4 Prediction Accuracy on New Data

We compared four different classifiers in order to see how well the clusters could be recreated. We preprocessed The Axis dataset and clustered using Fuzzy-C-Means. We then split it in a training set (70%) and a test set (30%). The model trained on the training set and predicted the cluster label on the test set and calculated an accuracy. For each classifier, the splitting, training, predicting and calculation of accuracy was done for 50 iterations and an average accuracy for each classifier was obtained. Those accuracies are shown in Table 4.5.

Figure 4.13 shows the Precision-Recall curves for all classifiers (from one iteration). A larger area under the curve means a higher accuracy. All classifiers have very similar curves and therefore have very similar tradeoff between precision and recall for different thresholds.

	K-Nearest Neighbors	Random Forest	XGBoost	LightGBM
Accuracy (%)	98.2	98.2	98.3	98.4

Table 4.5: Accuracies for the different classifiers.



(a) K-Nearest Neighbors

(b) Random Forest

(c) XGBoost

(d) LightGBM

Figure 4.13: Precision-Recall curves for four different classifiers.

4.5 Programming Language Comparison

To analyse the impact of programming language selection to our model, we compared five different programming languages:

- **Java**. 52837 samples. All samples in the *Technical Debt Dataset*.
- **Go**. 611 samples. Axis dataset.
- **C**. 1811 samples. Axis dataset.
- **TypeScript**. 515 samples. Axis dataset.
- **JavaScript**. 83 samples. Axis dataset.

In order to get an understanding of the differences in feature values between programming languages, we calculated both the mean and median values per language and feature. The result is shown in Tables 4.6 and 4.7 respectively.

We normalized each dataset (containing only commits in one programming language) and clustered using Fuzzy-C-Means with a membership threshold of 0.4 and calculated the mean value of each feature in the safe and fault-prone clusters separately. Table 4.8 shows the mean values for the safe cluster and Table 4.9 shows the mean values of the fault-prone cluster.

Feature	Java	Go	C	Typescript	JavaScript
LOC-ADDED	134.12	55.98	54.91	74.05	39.02
LOC-DELETED	44.33	18.68	40.50	30.95	16.42
MAX-LOC-WORKED-ON	74.02	45.95	63.98	49.36	34.92
AVG-LOC-WORKED-ON	35.49	28.72	42.07	21.46	24.43
TOTAL-CHANGED-FILES	4.85	2.46	2.08	4.21	1.63
MAX-COMMITS-TO-FILE	20.35	17.12	10.74	11.47	3.76
AVG-COMMITS-TO-FILE	12.91	12.48	7.63	7.59	3.61
MAX-AUTHORS	4.44	4.46	4.07	4.22	2.33
AVG-AUTHORS	3.41	3.83	3.03	3.33	2.22

Table 4.6: The mean value of the features before normalization for different datasets.

Feature	Java	Go	C	Typescript	JavaScript
LOC-ADDED	24.0	14.0	15.0	29.0	6.0
LOC-DELETED	7.0	3.0	6.0	10.0	1.0
MAX-LOC-WORKED-ON	26.0	18.0	24.0	24.0	10.0
AVG-LOC-WORKED-ON	17.0	14.0	18.0	15.5	6.5
TOTAL-CHANGED-FILES	2.0	1.0	1.0	3.0	1.0
MAX-COMMITS-TO-FILE	11.0	13.0	4.0	10.0	2.0
AVG-COMMITS-TO-FILE	8.0	9.0	3.29	7.0	2.0
MAX-AUTHORS	3.0	4.0	2.0	4.0	2.0
AVG-AUTHORS	2.74	3.89	2.0	3.0	2.0

Table 4.7: The median value of the features before normalization for different datasets.

Feature	Java	Go	C	Typescript	JavaScript
LOC-ADDED	-0.19	-0.24	-0.35	-0.56	-0.04
LOC-DELETED	-0.19	-0.22	-0.16	-0.46	-0.23
MAX-LOC-WORKED-ON	-0.22	-0.19	-0.37	-0.61	-0.10
AVG-LOC-WORKED-ON	-0.16	-0.02	-0.37	-0.58	-0.12
TOTAL-CHANGED-FILES	-0.20	-0.29	-0.24	-0.50	-0.04
MAX-COMMITS-TO-FILE	-0.49	-0.73	-0.23	-0.04	-0.59
AVG-COMMITS-TO-FILE	-0.46	-0.70	-0.21	0.23	-0.56
MAX-AUTHORS	-0.52	-0.78	-0.22	0.01	-0.57
AVG-AUTHORS	-0.49	-0.70	-0.20	0.25	-0.56

Table 4.8: The mean value of the normalized features in the safe cluster for different datasets.

Feature	Java	Go	C	Typescript	JavaScript
LOC-ADDED	0.31	0.23	0.99	0.79	0.08
LOC-DELETED	0.31	0.21	0.44	0.65	0.42
MAX-LOC-WORKED-ON	0.36	0.18	1.03	0.87	0.18
AVG-LOC-WORKED-ON	0.26	0.02	1.03	0.82	0.22
TOTAL-CHANGED-FILES	0.32	0.28	0.66	0.71	0.08
MAX-COMMITS-TO-FILE	0.79	0.71	0.64	0.05	1.09
AVG-COMMITS-TO-FILE	0.74	0.68	0.59	-0.33	1.05
MAX-AUTHORS	0.83	0.75	0.61	-0.01	1.05
AVG-AUTHORS	0.78	0.68	0.55	-0.36	1.04

Table 4.9: The mean value of the normalized features in the fault-prone cluster for different datasets.

Chapter 5
Discussion

5.1 Analysis of Results

In this section, we explain and discuss the importance of the results presented in Chapter 4.

5.1.1 Choice of Clustering Algorithm

K-Means

We investigated K-Means for both two and three clusters. The precision presented in Table 4.1 shows very high values for K-Means (2), especially in the interesting 90/10 distribution column. But the recall for the same model was very low, also for the 99/1 test, as seen in Table 4.2.

The K-Means with three clusters performed fairly well on the 90/10 and 99/1 precision tests. But for 90/10, the recall is approximately 13 percentage points below the best performing model in the column, Fuzzy-C-Means, and for 99/1 the model is about 10 percentage points below.

DBSCAN

As seen in Table 4.1, we can achieve comparatively high precision when using DBSCAN for the 90/10 and 99/1 tests. That might be since DBSCAN is good at finding outliers in datasets and most of our faulty commits are located at the less densely areas in the plot. Table 4.2 on the other hand shows poor recall measurements for the same tests. For larger proportion of fault-inducing commits, the recall is very good, but those proportions are not very likely to be found in reality.

One flaw using DBSCAN is that different datasets create various clusters. With the same DBSCAN model being executed on both the Axis dataset and the *Technical Debt Dataset*, the

clusters were not similar. More than twice as many commits go in the fault-prone cluster with Axis dataset compared to the *Technical Debt Dataset*. This shows that DBSCAN can not cluster two datasets of very different sizes in a similar manner.

Another disadvantage with DBSCAN is the tuning of hyperparameters. When working with the *Technical Debt Dataset*, we noticed that very small changes led to big changes on the performance. Since the Axis dataset differs a bit from the *Technical Debt Dataset*, it probably means that the same hyperparameters are not optimal for both datasets.

It is also a drawback to use DBSCAN since the parameters epsilon and min_samples most likely would need to be changed in a near future. As time goes on, the Axis dataset is growing (more commits are coming in), which means more samples. The parameters are dependent on the dataset and therefore they should be tuned every time the model is trained. With a growing dataset, one would probably need to decrease the epsilon or increase the min_samples more frequently than desired.

Agglomerative Clustering

Table 4.1 shows that the Agglomerative clustering algorithm is performing well at the 90/10 test, but not on the 99/1 test, where it gets the lowest precision. Agglomerative clustering performs the worst out of all tested clustering methods when it comes to recall, this is shown in Table 4.2.

Another flaw is that the Agglomerative clustering algorithm is very time consuming. The goal of our model is that it is supposed to work with any dataset, regardless of size. But when the dataset is growing bigger, it will take a significantly longer time to train the model compared to the other methods.

Gaussian Mixture

Gaussian mixture had the lowest precision out of all algorithms for the 90/10 distribution and the second worst precision for the 99/1 distribution, see Table 4.1. The algorithm did however, achieve a good recall with the second best recall for both the 90/10 distribution and the 99/1 distribution.

The reasoning behind Gaussian mixture being worse in both recall and precision compared to Fuzzy-C-Means might come from the fact that the fault-prone cluster fits better to a spherical shape than an elliptical.

The two major differences between Gaussian mixture and Fuzzy-C-Means are the elliptical shaped clusters instead of spherical clusters and the fact that the membership function works differently. Since we put more focus on optimizing membership thresholds for Fuzzy-C-Means than we did for Gaussian mixture, the worse performance might also stem from this. We encourage future work of analysing different membership thresholds for Gaussian mixture to get a complete understanding of the possibilities to use Gaussian mixture as a labeling technique in SFP.

Fuzzy-C-Means

With acceptable precision for both the 90/10 and the 99/1 distributions and with the best recall for the same distributions, we found Fuzzy-C-Means to be the overall best algorithm to use.

Since Fuzzy-C-Means was the best algorithm for the distributions we assumed most relevant, we decided to use it as the clustering technique of our model. After we had chosen Fuzzy-C-Means as the clustering algorithm for our model, all future analysis was done with Fuzzy-C-Means.

To get a comparable performance metrics to previously known models, we also calculated the AUC score of Fuzzy-C-Means using the *Technical Debt Dataset*, the ROC curve can be seen in Figure 4.8. As presented in Section 4.1.5, we achieved an AUC score of 0.6800 when using Fuzzy-C-Means. We also calculated a False Positive Rate of 38.1% and a False Negative Rate of 25.4%. Compared to previously known SFP models, this is a promising result.

To further validate that our clusters were representative of safe and fault-prone commits, we validated the Fuzzy-C-Means clusters against Jira. The difference between the number of tickets and commits in the respective clusters for all three components can be seen in Figures 4.9, 4.10 and 4.11.

To compare the different clusters to the tickets in Jira, we calculated two different distance metrics on the created curves. We used the first metric, dynamic time warp distance, since we expected a time shift from a commit being pushed in a project to the time an issue ticket was published in Jira. The DTW distance between the two clusters and Jira issues are presented in Table 4.3.

Another distance metric we used, partial curve mapping, also compared the similarity between the clusters and Jira issues. The results from this metric are presented in Table 4.4.

The results from both of the distance metrics indicated that our labeling by using Fuzzy-C-Means clustering closely resembled a labeling, where each issue in Jira was connected to a fault-prone commit. However, since we normalized the curves before comparison, the distances did not capture differences in the total amount of commits in each cluster and the total amount of issues in Jira. If our goal would have been to predict if each commit induces a fault or not, this would have been an issue but since we only were interested in if a commit was fault-prone and needed testing, we did not have to take the loss of information when normalizing in to consideration when validating our clusters.

5.1.2 Supervised Classifier Performance

From Table 4.5 and Figure 4.13, it can be seen that the four classifiers performed similarly (just above 98% accuracy). The accuracy was slightly better for the LightGBM classifier, and in addition to that, classifiers of this type are normally very fast.

Since the classifier did not manage to classify all samples perfectly with 100% accuracy, it was desired to see which samples that were misclassified. Figure 5.1 shows an enhanced scatter plot of the samples predicted with LightGBM (on the Axis dataset). The green dots are correctly classified as safe and the red dots are correctly classified as fault-prone. The blue x:es show the samples that the model predicted as fault-prone, but actually were safe. The black triangles show which samples were predicted safe while actually being fault-prone.

The dataset used for prediction contained 909 samples, 9 x:es and 4 triangles. All those 13 misclassified samples can be seen in this enhanced image because they are all located at the border between the two clusters. This means that we are not misclassifying any obvious samples as part of the wrong cluster. If the wrongly classified samples would be far from this border, it would indicate that the supervised machine learning model gave unreliable predictions.

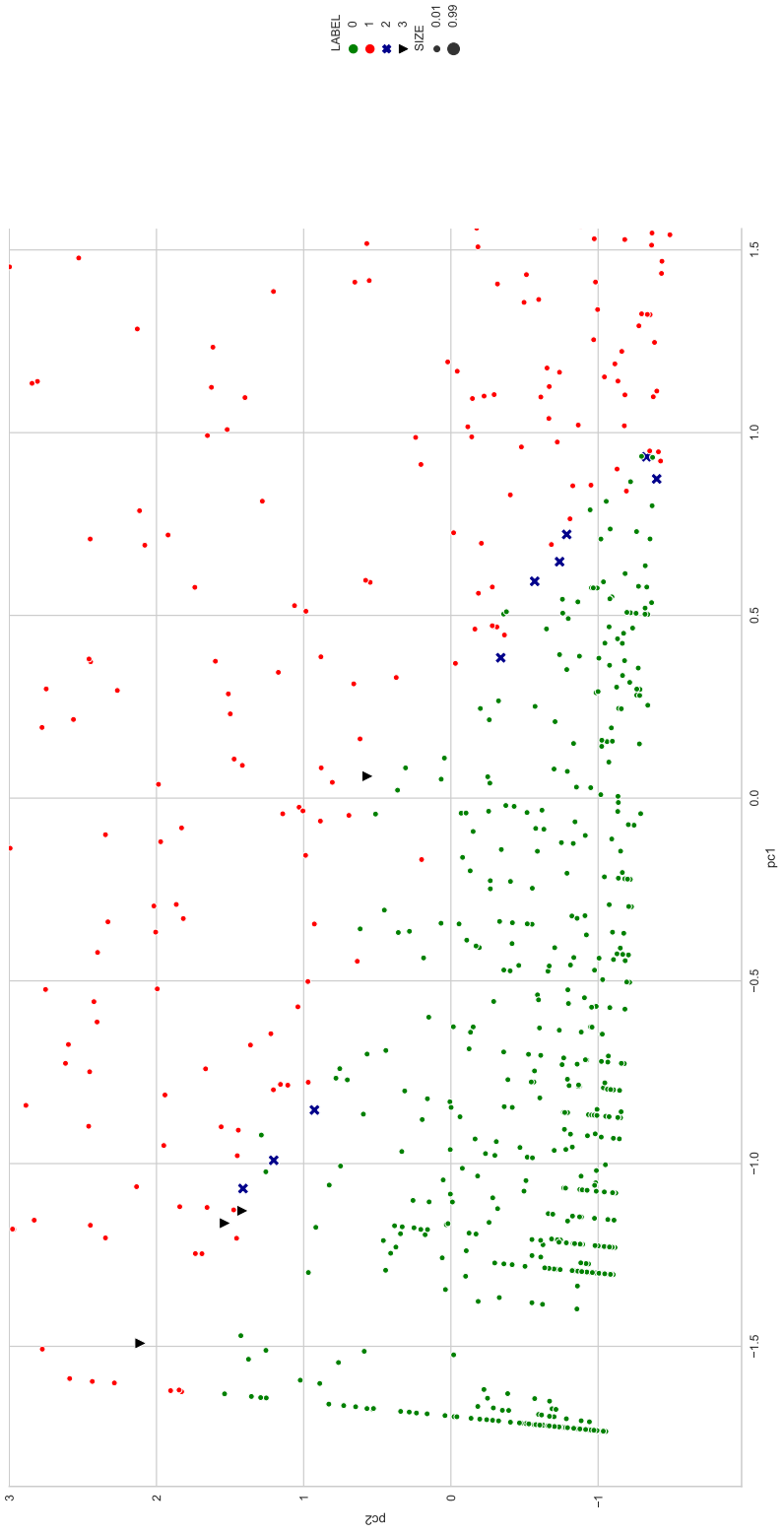


Figure 5.1: Scatter plot example of the misclassified samples.

5.2 Cross-project Application

To answer RQ2, we had to test our model on different types of data since large software systems contain multiple sub-projects with different teams working in different ways. As presented in Table 4.1 and 4.2, we tried our model on different distributions of safe and fault-prone commits by using random undersampling techniques on the *Technical Debt Dataset*. We made an assumption that the 90/10 and 99/1 distributions, meaning 90% or 99% safe commits and 10% or 1% fault-prone commits, were the most relevant in almost all software projects. This assumption indicated that Fuzzy-C-Means was a fitting algorithm to use as it resulted in a high recall of fault-prone commits, something we wished to achieve.

Since Fuzzy-C-Means assigns a membership value to each sample, we also experimented with changing the membership threshold for when a sample were to be considered as a member of the fault-prone class. The results from this study are presented in Table 5.1. As mentioned we desired a high recall. However, achieving a high recall is not difficult, all it takes is to increase the number of samples in the fault-prone class. This effect can be seen in Table 5.1, where the lowest membership threshold 0.3 yields the highest recall. To counteract this effect, we had to take the precision and the distributions between the two classes in to consideration when choosing a threshold. With this in mind, we ended up choosing 0.4 as the threshold to use since it achieved a recall over 70% while the precision for a fault-prone commit remained close to 2% for the 99/1 distribution and 20% for the 90/10 distribution. To strengthen our decision we also calculated F_2 scores for each threshold since we found recall more important than precision for our problem. A threshold of 0.4 gave comparably good F_2 scores for both distributions.

Threshold	Precision 90/10	Recall 90/10	F_2 Score 90/10	Precision 99/1	Recall 99/1	F_2 Score 99/1
0.3	0.17048	**0.77144**	0.45245	0.01731	**0.78525**	0.07953
0.35	0.17942	0.75041	**0.45855**	0.01813	0.76443	0.08279
0.4	0.18567	0.71443	0.45517	0.01921	0.74649	0.08708
0.45	0.19348	0.67319	0.45003	0.01989	0.69773	0.08927
0.5	**0.20030**	0.61979	0.43682	**0.02028**	0.62680	**0.08978**
0.55	0.19381	0.49432	0.37731	0.01813	0.47907	0.07873
0.6	0.18765	0.39855	0.32540	0.01756	0.40298	0.07476

Table 5.1: Precision, recall and F_2 score for different Fuzzy-C-Means decision thresholds.

5.3 Threats against Validity

In this section, we discuss threats that might affect the validity of our results.

5.3.1 Programming Language Selection

A comparison of the mean and median values in Tables 4.6 and 4.7 shows that JavaScript seems to have smaller values for all features. The small values on the last four features are probably a consequence of the dataset for JavaScript being very small: there is very little

history of the files. The TOTAL-CHANGED-FILES mean was also small for the dataset since few files actually were written in JavaScript.

Java typically has a bit larger code changes than the rest, which might have something to do with the dataset not being collected from Axis. Otherwise, the mean and median do not seem to differ by too much.

One more interesting thing to investigate is the mean values of features in the safe and fault-prone cluster respectively (Tables 4.8 and 4.9). Preferably the values in the fault-prone cluster should be larger than the feature values in the safe cluster. For all programming languages except TypeScript, every feature had a negative mean value in the safe cluster and a positive mean value in the fault-prone cluster. This result was to be expected and further validated our hypothesis of what the clusters represented.

For TypeScript on the other hand, the last three features had positive feature mean values in the safe cluster and negative in the fault-prone cluster. An explanation of this might be that TypeScript has very large differences in code change features (five first) between the safe and fault-prone cluster. The Fuzzy-C-Means then clusters the data based more on those features. This result indicates that the chosen clustering method might not be optimal for a dataset containing only code written in TypeScript.

5.3.2 The SZZ Algorithm and The *Technical Debt Dataset*

Validation of our clustering model was mainly based on the *Technical Debt Dataset*. The mining of the *Technical Debt Dataset* took advantage of the SZZ algorithm mentioned in Section 2.1.6. The authors of the *Technical Debt Dataset* paper (Lenarduzzi et al., 2019) used their own implementation of the SZZ algorithm and mentions some limitations with the algorithm.

For instance, the SZZ algorithm is based on commit messages to identify fault-fixing commits. Without standardized commit messages or unclear languages, some commit messages may be interpreted in the wrong way. Another issue is the limitation with a line-based diff provided by Git. The limitation can cause the algorithm to wrongfully identify the fix or a change in data-structures. Lastly, the algorithm does not consider fixes that may occur on another repository than the introduction of the fault. So if the fault is induced in the system in one repository and it is fixed in another repository, the SZZ algorithm would not be able to find the fault-inducing commit.

Since these issues were taken in to consideration when the *Technical Debt Dataset* was created and measures were taken to limit their impact, we decided to still use the *Technical Debt Dataset* to validate our model.

5.3.3 Inconsistency in Jira Issues

The second method we used to validate our results were similarity measures between clusters and Jira issues over time. To perform this validation we collected and filtered data from Jira with API queries. Since this validation method was performed on the Axis dataset we were interested in the difference between components. The “component” filter in Jira was used to separate the data in to three different datasets, one for each component.

However, by using this methodology we assumed that the “component” filter was used and that only the component that the issue actually was found in was mentioned in “component”. This assumption can be seen as a threat against validity since the component mentioned in the “component” field might not be the component from where the issue originated.

Since, in many cases, the origin of the issues is not obvious, the “component” field might be empty. We discarded these cases when the data was gathered from Jira. Since we collected and clustered all commits while we discarded some Jira issues, some time intervals might have had slightly different similarity scores than if we had not discarded any Jira issues at all.

Chapter 6
Conclusions

Software quality assurance has seen great development in recent years and machine learning has been in the forefront of the development. A popular machine learning tool used in SQA is software fault prediction, the ability to predict fault-proneness of a system based on the history of the system. In this thesis, we presented a novel software fault predictor based on clustering techniques and change metrics.

To answer RQ1, it can be seen that the AUC score of 0.68 as well as an FPR of 38.1% and an FNR of 25.4% our model is comparable with results from other models that are also based on unsupervised learning but use other features. Nam and Kim (2015) got an AUC score of 0.723 and Zhang et al. (2016) got 0.71. Catal et al. (2010) achieved an FPR of 34.55% and FNR of 25%. Our results show slightly worse performance than previously known models that are based on software metrics. The gathering of software metrics, however, is a tedious process and has to be done frequently while our model collects data in a more efficient way and does not suffer from the requirement of needing to be trained as often. So their small improvement of performance comes at a cost of time and resources.

To create a versatile fault-proneness analysis tool and to answer RQ2, the selected features are independent of project and programming languages as long as Git is used as version control system. However, training a model on one project and predicting on another might lead to worse performance since projects often differ a lot in programming languages and coding standards. Therefore it is advised to only use clustering techniques as fault-proneness prediction when working on one or few similar projects.

The answer to RQ3 is in our case the Fuzzy-C-Means clustering model. But this decision is based on that many people are experienced developers and coding standards are high. Limiting the number of programming languages or using only similar languages is therefore sufficient for our model. If there were a larger distribution of faults, another clustering technique might be superior.

Overall, the future of unsupervised software fault prediction is bright and promising results have been presented.

6.1 Future Work

The goal of this thesis is to lay the groundwork for an unsupervised software fault predictor based on change metrics. Since this type of model is relatively unexplored, it can be further analysed in great detail in the future. Some of our suggestions of future work are as follows:

- More extensive analysis of the effects of programming language selection.
- Analyse the idea of combining multiple clustering techniques.
- Measure performance of multiple clusters (more than three) for each algorithm. This was tried during the thesis work but since the clusters were hard to interpret for different datasets, no in depth analysis was conducted and further work is encouraged.
- Analyse different membership thresholds for Gaussian Mixture Models.
- A more in depth analysis of the supervised part of the model.
- Instead of classifying a commit either fault-prone or safe, a probabilistic fault-proneness estimation can be explored.

References

Abdeen, H., Bali, K., Sahraoui, H., and Dufour, B. (2015). Learning dependency-based change impact predictors using independent change histories. In *Information and Software Technology*, volume 67.

Bezdek, J. C., Ehrlich, R., and Full, W. (1984). FCM: The fuzzy c-means clustering algorithm. *Computers & geosciences*, 10(2-3):191–203.

Boucher, A. and Badri, M. (2018). Software metrics thresholds calculation techniques to predict fault-proneness: An empirical comparison. In *Information and Software Technology*, volume 96.

Brems, M. (2017). A one-stop shop for principal component analysis. `https://towardsdatascience.com/a-one-stop-shop-for-principal-component-analysis-5582fb7e0a9c`. [Online; accessed 17-February-2021].

Catal, C., Sevim, U., and Diri, B. (2010). Metrics-driven software quality prediction without prior fault data. In *Electronic Engineering and Computing Technology*. Springer.

Caulo, M. and Scanniello, G. (2020). A taxonomy of metrics for software fault prediction. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 429–436.

Choudharya, G. R., Kumar, S., Kumar, K., Mishra, A., and Catal, C. (2018). Empirical analysis of change metrics for software fault prediction. In *Computers and Electrical Engineering*, volume 67.

Dias, M. L. D. (2019). fuzzy-c-means: An implementation of fuzzy c-means clustering algorithm.

Fawcett, T. (2006). An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874. ROC Analysis in Pattern Recognition.

Grundy, J., Tekinerdogan, B., Ali, N., Mistrík, I., and Soley, R. M. (2015). *Software Quality Assurance.*

Jekel, C. F., Venter, G., Venter, M. P., Stander, N., and Haftka, R. T. (2019). Similarity measures for identifying material parameters from hysteresis loops using inverse analysis. *International Journal of Material Forming*, 12.

Kumar, S. and Rathore, S. S. (2018). *Software Fault Prediction: A Road Map*. SpringerBriefs in Computer Science. Springer Singapore.

Kyaagba, S. (2018). Dynamic time warping with time series. `https://medium.com/@shachiakyaagba_41915/dynamic-time-warping-with-time-series-1f5c05fb8950`. [Online; accessed 24-February-2021].

Lenarduzzi, V., Saarimäki, N., and Taibi, D. (2019). The technical debt dataset. In *The Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE'19)*.

Li, Z., Jing, X.-Y., and Zhu, X. (2018). Progress on approaches to software defect prediction. *IET Software*, 12(3):161–175.

Liu, J., Zhou, Y., Yang, Y., Lu, H., and Xu, B. (2017). Code churn: A neglected metric in effort-aware just-in-time defect prediction. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 11–19.

Mishra, S. (2017). Unsupervised learning and data clustering. `https://towardsdatascience.com/unsupervised-learning-and-data-clustering-eeecb78b422a`. [Online; accessed 10-February-2021].

Nam, J. and Kim, S. (2015). Clami: Defect prediction on unlabeled datasets (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 452–463.

Pathak, M. (2018a). Hierarchical clustering in R. `https://www.datacamp.com/community/tutorials/hierarchical-clustering-R`. [Online; accessed 04-March-2021].

Pathak, M. (2018b). Introduction to t-SNE. `https://www.datacamp.com/community/tutorials/introduction-t-sne`. [Online; accessed 17-February-2021].

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

Sharma, A. (2020). How to master the popular DBSCAN clustering algorithm for machine learning. `https://www.analyticsvidhya.com/blog/2020/09/how-dbscan-clustering-works/`. [Online; accessed 04-March-2021].

Sharma, P. (2019). The most comprehensive guide to k-means clustering you'll ever need. https://www.analyticsvidhya.com/blog/2019/08/comprehensive-guide-k-means-clustering/. [Online; accessed 10-February-2021].

Sing, A. (2019). Build better and accurate clusters with gaussian mixture models. *Analytics Vidhya*, 1.

Xu, Z., Li, L., Yan, M., Liu, J., Luo, X., Grundy, J., Zhang, Y., and Zhang, X. (2021). A comprehensive comparative study of clustering-based unsupervised defect prediction models. *Journal of Systems and Software*, 172.

Yang, B., Zheng, X., and Guo, P. (2006). Software metrics data clustering for quality prediction. In Huang, D.-S., Li, K., and Irwin, G. W., editors, *Computational Intelligence*, pages 959–964. Springer Berlin Heidelberg.

Yiu, T. (2019). The curse of dimensionality. https://towardsdatascience.com/the-curse-of-dimensionality-50dc6e49aa1e. [Online; accessed 17-February-2021].

Zhang, F., Zheng, Q., Zou, Y., and Hassan, A. E. (2016). Cross-project defect prediction using a connectivity-based unsupervised classifier. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 309–320.

Zhang, H., Si, S., and Hsieh, C.-J. (2017). GPU-acceleration for large-scale tree boosting.

Zhang, Z. (2019). Understand data normalization in machine learning. https://towardsdatascience.com/understand-data-normalization-in-machine-learning-8ff3062101f0. [Online; accessed 20-April-2021].

Śliwerski, J., Zimmermann, T., and Zeller, A. (2005). When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, page 1–5. Association for Computing Machinery.

Appendices

Appendix A
Decision Tree

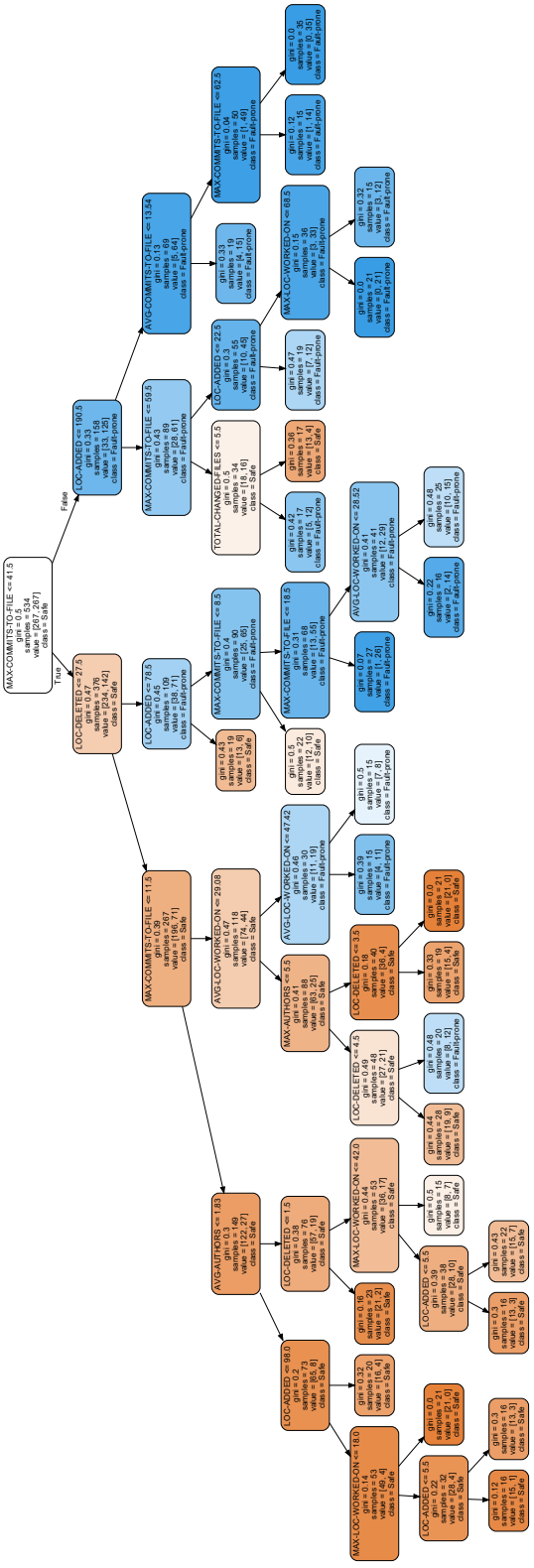


Figure A.1: An example of a decision tree created with a decision tree classifier.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2021-06-10

EXAMENSARBETE Unsupervised software fault prediction using change metrics
STUDENTER Oskar Holmqvist, Elias Tedenvall
HANDLEDARE Pierre Nugues (LTH), Henrik Forsberg (Axis), Kenneth Nilsson (Axis)
EXAMINATOR Martin Höst (LTH)

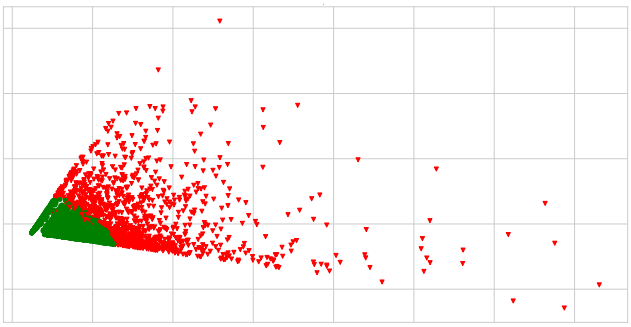
Oövervakad felprediktion av mjukvara med förändringsfaktorer

POPULÄRVETENSKAPLIG SAMMANFATTNING **Oskar Holmqvist, Elias Tedenvall**

Kvalitetssäkring av mjukvara är en väldigt tidskrävande och kostsam process som är nödvändig i alla programvaruföretag. I detta arbete utforskas en oövervakad maskininlärningsteknik som utför en riskbedömning av kodbasen.

Det är extremt viktigt för mjukvaruföretag att ha fungerande program som inte uppför sig oväntat eller innehåller buggar. Exempelvis skulle det vara katastrofalt om en övervakningskamera i en bank slutade fungera mitt under ett rån. För att förhindra olika typer av fel i mjukvara så jobbar företag med så kallad kvalitetssäkring. Det innebär att man kontinuerligt testar programmen och ser till att allt fungerar som det ska. Tyvärr har de flesta företag inte tillräckligt med tid eller resurser för att frekvent kunna testa alla delar av systemet. Det innebär att företagen måste prioritera att testa de delar av programvaran som har störst risk att innehålla felaktigheter.

Vi har i detta examensarbete undersökt hur väl olika algoritmer som använder oövervakad maskininlärning kan prediktera om en förändring i koden introducerar defekter eller inte. En förändring innebär att en utvecklare har lagt till, tagit bort eller modifierat kodrader. Oövervakad maskininlärning innebär att modellen inte känner till tidigare defekta förändringar vid träning. Algoritmerna grupperar förändringarna så att stora förändringar, som antas ha högre risk, hamnar i en grupp, medan små förändringar klassificeras att ha låg risk och hamnar i en annan grupp. I figuren representeras små förändringar (låg risk) av gröna prickar medan stora förändringar (hög risk) är representerade av röda trianglar. Med denna riskbedömning ska företag prioritera att testa de delarna av systemet där det skett många högriskförändringar under den senaste tidsperioden.



Resultaten visar att det finns ett samband mellan förändringar som vår modell klassificerar som riskfyllda med faktiska defekter i programvaran. Det kunde vi se genom att validera modellen med data från en kodbas där det redan var känt vilka förändringar som introducerade felen. Modellen är dessutom väldigt generell och kan tillämpas på många olika projekt utan större justeringar. I jämförelse med existerande modeller så är vår modell generellt sett snabbare och ger likvärdig tillförlitlighet.