MASTER'S THESIS | LUND UNIVERSITY 2018

Clustering and Classification of Test Failures Using Machine Learning

Andy Truong, Daniel Hellström

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2018-30

Clustering and Classification of Test Failures Using Machine Learning

Andy Truong
andychiwi.truong@gmail.com

Daniel Hellström
d.hellstrom.94@gmail.com

June 5, 2018

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

Supervisor: Erik Larsson, erik.larsson@eit.lth.se

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

Verification is an important part of the hardware design process. Thorough verification requires a large number of test cases. To facilitate debugging in verification, test failures are clustered and classified according to their root causes. Currently, this is performed by manually examining log files. This thesis investigated how machine learning can be applied to automate this process.

We used Python and Scikit-learn to evaluate and compare machine learning algorithms for clustering and classification of test failures. Our dataset consisted of 12500 log files with 12 different root causes. We compared eight classification algorithms and three clustering algorithms. We also investigated how dimensionality reduction affected both computation time and performance. The most suitable clustering algorithm was DBSCAN with AMI and ARI scores of 0.593 and 0.545 respectively. The most suitable classification algorithm was random forest with an accuracy of 0.907 and an F_1-score of 0.913.

Keywords: machine learning, hardware verification, simulation, log file, classification

Acknowledgements

First, we would like to thank Lars Viklund and Harry Duque at Axis Communications for their continuous support of this thesis. Their insight and guidance helped us in our investigation. We would also like to show our gratitude towards Harry and the verification team at Axis for providing us with an annotated dataset. Our thesis would not have been possible without it.

We would also like to thank our supervisor at Lunds Tekniska Högskola, Erik Larsson, for feedback during this thesis. Special thanks for helping us with the academic writing process.

Contents

List of abbreviations

AMI	Adjusted mutual information
ARI	Adjusted Rand index
CRV	Constrained random verification
DBSCAN	Density-based spatial clustering of applications with noise
DUT	Device under test
HDL	Hardware description language
IC	Integrated circuit
MDS	Multidimensional scaling
PCA	Principal component analysis
RBF	Radial basis function
SFM RF	SelectFromModel using random forest
SFM SVC	SelectFromModel using support vector classifier
SVM	Support vector machine
t-SNE	t-distributed stochastic neighbour embedding
UVC	UVM verification component
UVM	Universal verification methodology

Chapter 1
Introduction

Discovering design flaws during manufacturing of integrated circuits (ICs) is very costly. Pre-silicon verification is used to discover design flaws before IC manufacturing. In industry, pre-silicon verification is typically performed using simulation. In simulation, a test bench applies a number of test cases to the design and the produced responses are compared against expected responses, the reference. If test cases fail, a root cause analysis takes place in order to determine the root causes of the failures. The root cause analysis process consists of two steps, *clustering* and *classification*. The first step is clustering the test failures. The goal of clustering is to determine which test failures share the same root cause. After the test failures have been clustered, a classification is performed to determine the type of root cause for each failure.

Successful root cause analysis facilitates efficient distribution of debugging efforts. The process of designing hardware consists of strict deadlines. Assigning verification engineers to root cause analysis is dependent on the deadlines and more resources are necessary for debugging if the deadline is closer. An efficient distribution includes the determination of which and how many engineers should be assigned the task of debugging. By knowing the root causes of test failures, they can be assigned to the engineers best suited to correct them [1].

Root cause analysis is often performed manually by examining log files, which can be tedious and time consuming. According to a study performed in 2014, 37% of a verification engineer's time is spent on debugging [2], which root cause analysis is part of. Reducing the time and effort required for root cause analysis, and in turn debugging, would have a positive impact on the entire verification process [3].

One method of reducing the time spent on root cause analysis is by automating the process. Automation of root cause analysis can be achieved using different approaches. Rule-based approaches seek to create a set of rules that correlate events with root causes. Model-based approaches construct an approximate model of the system. These approaches require extensive knowledge of the system and such knowledge can be difficult to obtain. Another alternative is to use a machine learning approach. This approach does not require

extensive knowledge about the system. Instead, machine learning relies on large amounts of data to model relationships between test failure and root cause [4].

Determining how effective machine learning techniques are for a problem is difficult without empirical results. There are several different algorithms used for clustering and classification. The algorithms use different strategies which leads to them performing differently depending on the problem. In this thesis we will investigate how machine learning can be used to automate clustering and classification of test failures. We will compare algorithms for both problems in order to determine the algorithms that are best suited for automation of clustering and classification of test failures.

1.1 Goal

The goal of the thesis is to investigate how machine learning techniques can be applied to automate the process of clustering and classification of test failures. The main questions are:

- How can machine learning techniques be applied effectively to root cause analysis of IC designs?
- Which type of machine learning algorithm is the most suitable for root cause analysis?

1.2 Related Work

In this section, the results of related work and the difference compared to this thesis will be described.

Chen et al. [5] investigated how a machine learning algorithm called decision tree could be applied to root cause analysis. Their results indicated that the decision tree was applicable to their specific task of root cause analysis. A decision tree is a machine learning algorithm utilising a tree structure for classification. However, no comparison to other algorithms was made, which we will investigate further.

Lal and Pahwa [6] investigated the classification of root causes of software failures using different machine learning algorithms. Their methodology consisted of processing free-text bug reports and a comparison of classification methods. They concluded that a machine learning algorithm called support vector machine, which will be described later, yielded the best results. However, their choice of classification methods was restricted to those capable of working with unlabelled data due to their limited amount of labelled data, i.e. failures with known root causes. Since we have access to a sufficiently large set of labelled data we have used more accurate methods that require a larger set of labelled data.

In two studies, Chakrabarty et al. investigated the effectiveness of decision trees [4] and support vector machines [7] in root cause classification at board-level. The results indicated that both algorithms can be effective in root cause analysis. However, the number of data samples used in their evaluation was limited to 1000. Thus, no measure of how the

algorithms will perform on larger sets of data samples was given. We have used a larger amount of data samples in our evaluation.

The application of clustering methods for root cause analysis of software failures was investigated by Podgurski et al. [8]. They concluded that clustering is best used in conjunction with a visualisation algorithm since the results can be unreliable. The method of combining clustering with visualisation aids the identification of minor errors in the results. This method has been adopted in our thesis, but we have evaluated and compared multiple algorithms. Additionally, their study did not provide any metrics for the quality of the clustering, instead conducting a qualitative analysis. In this thesis, we used clustering metrics to determine the quality of our results.

1.3 Method

Our method consisted of three steps: preparation, data preprocessing and evaluation. During the preparation step we gathered information about machine learning algorithms and about how log files could be used as data for these algorithms. We also discussed how this data would be generated and labelled with the verification engineers at Axis. The preparation step resulted in a selection of algorithms that would be evaluated for this problem, a method for generating the data and a list of root causes that would be used to label this data.

The data preprocessing step consisted of further examining the log files that would be used as data in order to develop a method for extracting relevant information from them. With the help of the verification engineers at Axis we were able to extract this information from the log files and transform it so it could be used as input for the machine learning algorithms.

When the data preprocessing method had been developed and a sufficiently large dataset of 125000 log files had been generated for us, we performed an evaluation of the machine learning algorithms.

1.4 Disposition

The disposition of this thesis is as follows: Chapters two and three will provide theoretical background of the pre-silicon hardware verification and machine learning areas. In chapter two we will introduce the verification techniques and environment examined in this thesis and further describe the problems we aim to solve. In addition to providing the fundamentals of machine learning, chapter three will introduce the clustering and classification algorithms evaluated in this thesis. Chapter four will describe the data we used and how we transformed it into a format that could be used in machine learning. Our evaluation process and the results of the evaluation will be presented in chapter five and discussed in chapter six. Finally, in chapter seven the conclusions of the thesis will be presented.

1.5 Contributions

During the background investigation we both investigated the same subjects and discussed our findings. The method and code used in the data preprocessing step was mainly developed by Daniel. He also implemented the tool described at the end of chapter five. Andy mainly worked on the code for the evaluation as well as performed it.

The main contributions for to the report were the following: In chapter two, Daniel wrote about the methods and test bench while Andy focused on the root cause analysis process and its problems.

In chapter three, Daniel wrote about the fundamentals and the features in machine learning. Daniel also wrote about the metrics for clustering while Andy focused on the metrics for classification. The machine learning algorithms Andy described were logistic regression, naive Bayes, decision tree, K-nearest neighbours, K-means and DBSCAN. The algorithms Daniel described were support vector machine, agglomerative clustering and the visualisation algorithms.

In chapter four, Andy wrote about the data and Daniel about how the data was transformed.

In chapter five, Andy wrote about the algorithm selection and optimisation steps of the evaluation of classification algorithms and Daniel wrote about the final evaluation of those algorithms. The section about the clustering evaluation was written mainly by Andy, and the section describing the tool implementation mainly by Daniel.

We discussed the results with each other and both contributed equally to the discussion and conclusion chapters.

Chapter 2
Pre-Silicon Hardware Verification

This chapter will provide an introduction to the pre-silicon hardware verification process. The verification methods and verification environment examined in this thesis will be presented. The process of clustering and classification of test failures will be described along with the problems that they present.

2.1 Background

Verification of hardware designs can be performed using different strategies. The strategy investigated in this thesis was simulation-based verification. In simulation, hardware designs are implemented in a *hardware description language* (HDL) and simulated in a *test bench*. A simulated hardware design, known as the *device under test* (DUT), is tested by exposing it to combinations of input signals, or *stimuli*, and verifying that the output is correct [1].

In simulation, a series of stimuli combinations and output verification is called a *test case*. Previously, test cases were generated manually by verification engineers. As the complexity of hardware designs increased it grew more difficult to create the required amount of test cases. To address this problem, engineers started to generate test cases with randomised stimuli [1]. Designs provide a set of *constraints* that limit the input signals to ones the DUT can handle. The method of generating randomised test cases under constraints is known as *constrained random verification* (CRV) [9]. In order to generate many test cases that seek to verify the same part of the DUT, CRV enables the construction of templates that can be used to generate tests with randomised stimuli. Thus a low amount of manual effort is required to create a large set of test cases, many of which will explore input combinations that could have been missed if the test cases had been created manually [1]. CRV is the method used to generate the test cases examined in this thesis.

A drawback of simulation is that it is difficult to determine if all aspects of the design are covered by the test cases. To solve this problem, coverage methods are used. Two com-

monly used coverage methods are *structural coverage* and *functional coverage*. Structural coverage measures how much of the design's HDL code is executed while functional coverage measures how much of the desired functionality has been tested. Measuring structural coverage is done automatically, but no such methods exist for functional coverage. Instead, verification engineers have to explicitly specify the functionality that needs to be measured in functional coverage models [1].

2.2 Test Bench

In this section, the test bench, which is the environment where designs are simulated and tested, will be presented. In addition to the DUT, a test bench typically contains components such as stimulus generators, monitors, checkers and scoreboards. A component in a test bench is called a UVM verification component (UVC) where UVM stands for universal verification methodology [10]. The components in a test bench are implemented independently meaning that the components can be reused in other projects and across different test benches. An overview of a simple test bench, as described by Wile et. al. [1], is presented in Figure 2.1 and the remainder of this section will describe the components and how they are connected.

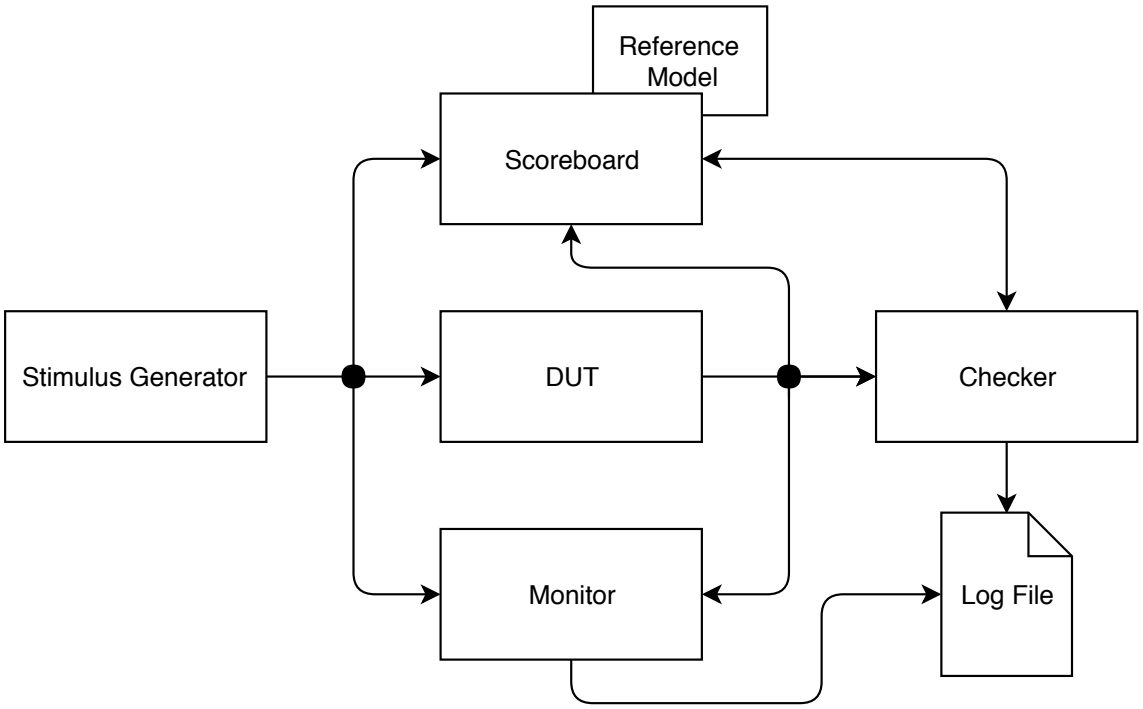

Figure 2.1: A simple test bench environment.

The first component is the stimulus generator. A stimulus generator initiates events by generating the stimuli used to test the DUT. In Figure 2.1, the stimuli generator provides input for the DUT, the monitor and the scoreboard components.

A monitor is a component that observes the inputs and outputs to and from the DUT, as seen in Figure 2.1, to verify that the protocol is followed. The protocol is derived from the

design specification and specifies valid responses and legal values. An error is returned by the monitor if the protocol is not followed. The monitor also observes the stimuli sent to the DUT and uses them to determine the functional coverage. The functionality to observe consists of internal states of the DUT and interesting cases specified in the functional coverage model. The monitor is also the component responsible for generating post-simulation information that can be used during debugging, which is recorded in a log file.

The scoreboard is responsible for containing the *reference model* and calculating the reference output that the output from the DUT can be compared to. The reference model is a high-level representation of the desired behaviour of the DUT. In Figure 2.1, the scoreboard is connected to both the stimulus generator and the DUT and observes the input stimuli and the function performed by the DUT. The scoreboard then uses the reference model to perform the same calculations as the DUT, which yields the reference output. When requested, the reference output will be sent to the checker where it is compared to the output from the DUT.

The checker is a special type of monitor. Its main job is to examine the output signals from the DUT, as shown in Figure 2.1, and determine if they are correct. When the checker observes output from the DUT it queries the scoreboard for the reference output and performs a comparison. If the checker observes output that does not correspond with the reference it records this in a log file so it can be used for debugging.

2.3 Root Cause Analysis

In this section, the importance and the process of root cause analysis will be described. The root cause analysis process consists of two steps. When simulated test cases fail, the first step is to cluster test failures that have the same root cause. The second step is to classify the root causes of the failures to decide how to proceed with debugging.

All projects are working with deadlines when developing new products. In the process of designing ICs, these deadlines are strict since it is difficult and expensive to make adjustments to IC designs after they are manufactured. Root cause analysis is important since it provides an overview of how many flaws exist and the severity of these flaws. A flaw in the design is considered as critical since discovering design flaws closer to manufacturing can be very costly and efforts need to be focused on correcting these flaws before the deadline. Flaws in the verification environment can be seen as not critical, if there is no risk of masking design flaws, and can be corrected after the deadlines. The time plan together with the severity of flaws are used to facilitate an efficient distribution of debugging efforts, in engineering cost and time. Too few resources will increase the time-to-market thus increasing the cost. Using too many resources results in over-investment. Therefore, it is important to determine the optimal amount of resources, which is facilitated by root cause analysis [1].

2.3.1 Root Cause Examples

The following paragraphs will provide examples of the root causes that were examined in this thesis. We investigated a set of the root causes provided to us by the verification engineers at Axis. The provided set consisted of common root causes in a set of different

test benches and designs. The investigated root causes can be summarised by the following groups:

- Failing checks or scoreboard mismatches between reference model and DUT due to fault in either check, design or reference model.
- Internal failures in the test bench infrastructure, e.g. components are not correctly connected.
- Illegal stimuli to the design under test caused by missing or incorrect constraints.
- Constraint solver errors caused by contradicting constraints.
- Constraint solver time-outs caused by constraints that are hard to solve.
- Incorrectly defined or configured functional coverage models.
- Intermittent failures caused by the IT infrastructure (full disk, network problems, etc.).

Scoreboard Mismatch

Scoreboard mismatches appear when the checker detects dissimilarities between the outputs from the DUT and the reference model. The root cause of a scoreboard mismatch can be located in the DUT, reference model or checker. When the root cause is located in the DUT or reference model it indicates the existence of a flaw that compromises the functionality. The reference model consist of components implemented in different programming languages such as OpenCL [11], SystemC [12] and Python [13]. A flaw in the implementation of the reference model results in test failures. Flaws located in the DUT are design flaws and should be sent to the design engineers so they can be corrected. Thus, determining the location of root causes of scoreboard mismatches is an important part of root cause analysis.

Connectivity

A flaw in the checker indicates that the values could be compared incorrectly or that there could exist a flaw in the checker since the components in the test bench are implemented independently. This results in the root cause of test failure to be classified as a UVC flaw. One consequence of implementing components independently is that the verification engineer is required to connect the components correctly in order to verify the design. For example, the functional coverage can not be determined if the monitor is not connected to the DUT. A faulty connectivity can also result in data not being transferred correctly between components.

Constraints

The usage of CRV requires the space of input stimuli to be limited by constraints. Examples of areas to constrain are the format of data and configurations of registers. Constraints defining a distribution function are also necessary since some values are more common.

The space of input stimuli in combination with distribution functions result in complex constraints. This complexity can lead to missing or incorrect constraints in the test bench which results in illegal stimuli to the design. There is also the risk of defining constraints that contradict each other, e.g. specifying that the value of a stimuli should be zero and one at the same time. Such contradictions will make the constraints impossible to satisfy.

In general, the problem of finding values that satisfy a set of constraints is NP-complete. By restricting the general problem it is however possible to efficiently solve it [14]. Thus, the constraints used in CRV need to be carefully implemented for the problem to remain tractable. Poorly implemented constraints can cause the time required to solve them to be longer than allowed, causing test cases to time-out.

Functional Coverage Models

As explained in Section 2.1, verification engineers have to specify the functionality that should be measured in functional coverage models. Thus, there is a risk of incorrectly defining the models, causing them to be inconsistent with the DUT or the test cases. For example, an incorrectly defined functional coverage model may designate certain stimuli, which are considered legal by the constraints, as illegal. These stimuli will then cause the monitor to raise a functional coverage error and fail the test.

Intermittent Failures

Intermittent failures caused by the IT infrastructure are not caused by how the verification engineers have implemented the reference model, the components in the test bench or the constraints for input stimuli. Examples of intermittent failures are running simulation on an environment with low memory resulting in full disk, a power shortage, the power has been unplugged or the network has temporarily gone down. Most intermittent failures can be resolved by simply rerunning the test cases.

2.3.2 Process and Problems

This section will provide an example of the clustering and classification process and also present the problems. The result of a clustering and classification process is presented in Figure 2.2. The figure visualises a hypothetical set of test failures using an arbitrary projection on a two dimensional space. Each shape in the figure represents a log file generated by a test failure and the distance between them indicates how similar they are. In the figure, clusters identified by root cause analysis are represented by the circles surrounding the test failures. The triangles and the squares represent two different types of root causes. In the example, three clusters were identified, indicating three different root causes. Root cause classification then reveals that the root causes are of two types: inconsistent constraint and reference model bug. By combining the results of clustering and classification it can be determined that there are two inconsistent constraints that manifest themselves differently, as well as one bug in the reference model. The results of root cause analysis are then used to determine the severity of the failures. Both the reference model bug and the inconsistent constraints are located in the test bench. Thus, these flaws are not as critical as a design flaw. With knowledge about the number of flaws and their severity, the debugging can be

divided into tasks and distributed efficiently among the engineers. The information from root cause analysis is used to determine the number of engineers and the time required to correct the flaws which lead to a lower verification resource cost.

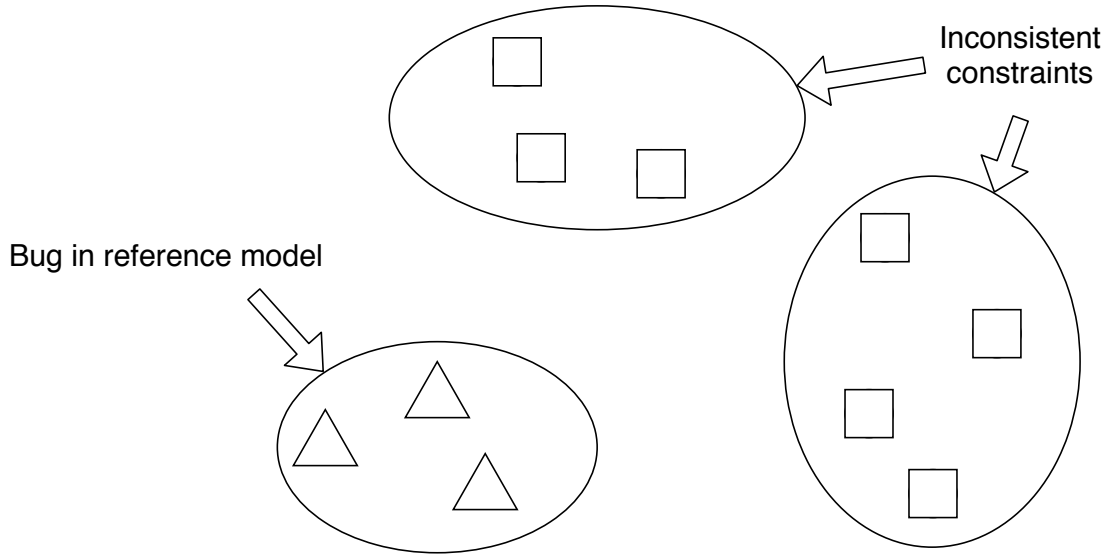

Figure 2.2: A clustering and classification of test failures visualised in two dimensions. Three clusters were identified with two types of root causes.

The main problem with clustering and classification is that they are performed by manually examining the log files generated during test executions. Log files can be large and verbose, making them difficult to process for humans [1]. The usage of CRV leads to test runs containing thousands of test cases. A test run could contain up to 8000 test cases and produce 100 test failures. A verification engineer is required to examine all 100 test failures when performing clustering and classification which is tedious and time-consuming. Reducing the time spent on clustering and classification of test failures is by automation. One way of automating of clustering and classification is to using machine learning, which will be introduced in the next chapter.

Chapter 3
Machine Learning

This chapter will provide a theoretical background for machine learning and important terms within the area will be introduced. The algorithms used in our evaluation will be presented and described. Finally, the methods and metrics that were used to evaluate the algorithms will be presented.

3.1 Fundamentals

The basic concept behind machine learning is the use of large *datasets* to teach computers about subjects and to be able to make decisions or uncover information. There are two main groups of machine learning: *supervised* and *unsupervised* learning. They both attempt to extract patterns from large amounts of data but operate in different ways.

Supervised learning uses *labelled* datasets to train *models* that can be used to predict outputs for new data. In a labelled dataset each object, or *sample*, has a label containing the output that should be associated with the sample. During training, different *algorithms* can be used to create a model of the relationships between samples and their labels. This model can then be used to determine the labels for new samples using the relationships it has learned [15].

Supervised learning can in turn be divided into two groups: *classification* and *regression*. Classification is used when the label can be one of multiple non-overlapping discrete classes. An example is a spam filter that has to determine if mails are spam or not. Regression is used when the label should be a continuous numerical value, such as when predicting the value of a stock [16]. In the case of classification, the model is often referred to as a *classifier*, which is the type of supervised learner used in this thesis.

Unsupervised learning does not require any labels for the input samples and does not train models. Instead, it relies on finding patterns in datasets without any external help. The most common type of unsupervised learning is clustering. Clustering algorithms seek to create clusters where the samples within a cluster are similar [15].

3.2 Features

To utilise machine learning algorithms, samples need to be represented by numerical values. The different numerical values representing a sample are called *features*. Some examples of features are the top-speed of a car or the frequency of the word "feature" in a text. Anything that can be converted into a numerical value can be a feature. Since the features are the only pieces of information machine learning algorithms will receive about the problem it is important that they carry relevant information. For example, to predict the temperature in a city on a specific date, the size of the city is irrelevant as a feature since it will remain constant regardless of the temperature. Including irrelevant features makes it more difficult for the algorithm to see the patterns present in the other features.

3.2.1 Dimensionality

For every new feature added to the feature set the dimension of the feature space increases. Many machine learning projects are using hundreds up to tens of thousands of features [17]. Therefore, the dimensionality of the spaces machine learning algorithms are supposed to find solutions in can become very large. One problem with high dimensionality is that the computation time of an algorithm increases. Another problem is that the more dimensions the feature space has the less likely two samples are to be close to each other, a phenomenon known as the *curse of dimensionality*. Géron [16] illustrates this with the following example: If two points are randomly placed in a square where each side has length 1, the average distance between them will be 0.52. When the same is attempted in a cube the average distance will be 0.66 [18]. If the dimensionality is large, a sample will often be far away from the other samples belonging to the same class or cluster. This increases the difficulty of forming clusters and classifying samples [16].

3.2.2 Dimensionality Reduction

Reducing the dimensionality of the feature set will help combat the curse of dimensionality while also reducing computation times. The reduction can be performed by using either *feature selection* or *feature extraction*. Feature selection methods reduce the dimensionality by removing irrelevant features, whereas feature extraction methods merge existing features [19].

There are two categories of feature selection methods used to determine the most relevant features: *filter* and *wrapper* approaches. Filter methods evaluate each feature separately by using a univariate scoring metric, such as variance, to determine the importance of the feature. Wrapper methods use different strategies to search for subsets of the features in order to find the subset which yields the highest accuracy when used with a classifier. An effect of this difference is that wrapper methods take the dependency between features into consideration while filter methods do not [20]. This typically leads to wrapper methods outperforming filter methods [21] at the cost of longer computation times [20].

Feature extraction can either be done manually or by using a feature extraction algorithm. Most of these algorithms are mathematical algorithms that represent data in a lower dimension. The most popular of these is *principal component analysis* (PCA). It finds the

hyperplane of the desired dimension that preserves the most information when the dataset is projected on it. PCA tends to preserve differences between samples well and is therefore well suited for preparing data before it is used in a machine learning algorithm [16].

3.3 Algorithms

In this section, the classification and clustering algorithms that were evaluated will be introduced. The evaluated classification algorithms are: logistic regression, support vector machine, naive Bayes, decision tree, random forest and K-nearest neighbours. For clustering, the evaluated algorithms are: K-means, DBSCAN and agglomerative clustering. The benefits of visualisation algorithms will also be described.

3.3.1 Classification

Classification algorithms are supervised algorithms that seek to train a classifier to accurately predict classes for new samples. Classifiers can be either *binary* or *multiclass*. Binary classifiers, for example logistic regression and support vector machines, only handle cases where there are two classes. In many cases, such as a spam filter, this is sufficient. However, many other problems require the output to be of more than two classes. Some classifiers, for example the random forest classifier, are capable of handling multiple classes. In order to not be limited to classifiers that are inherently multiclass, it is possible for a binary classifier to handle problems with multiple classes by using the *one versus all* approach. The one versus all approach consists of training multiple classifiers of the same type, one for each class, that can determine if a sample belongs to the class or not. The result is then the class that yields the best score [22].

Logistic Regression

Logistic regression is a classifier that was introduced 60 years ago by David Cox [23]. It is widely used in machine learning and medical fields. In the medical field, this model can be used for predicting mortality in hospitals [24] or predicting the risk of liver damage from medicines [25] based on observations of the patient. In engineering, this model can be used to predict the probability of system failures [26].

The logistic regression classifier outputs a number between 0 and 1 using a sigmoid function, as illustrated in Figure 3.1. These values are fitting for a classifier which describes a likelihood. The logistic regression classifier describes the probability of belonging to class 1 as minimal for lower x but when reaching a threshold, typically 0.5, the probability rises dramatically and stays close to 1 for larger x. These two properties are the reasons why the logistic regression classifier is popular [27]. However, this model is not accurate when there is noise in the data and these situations are common in systems [28].

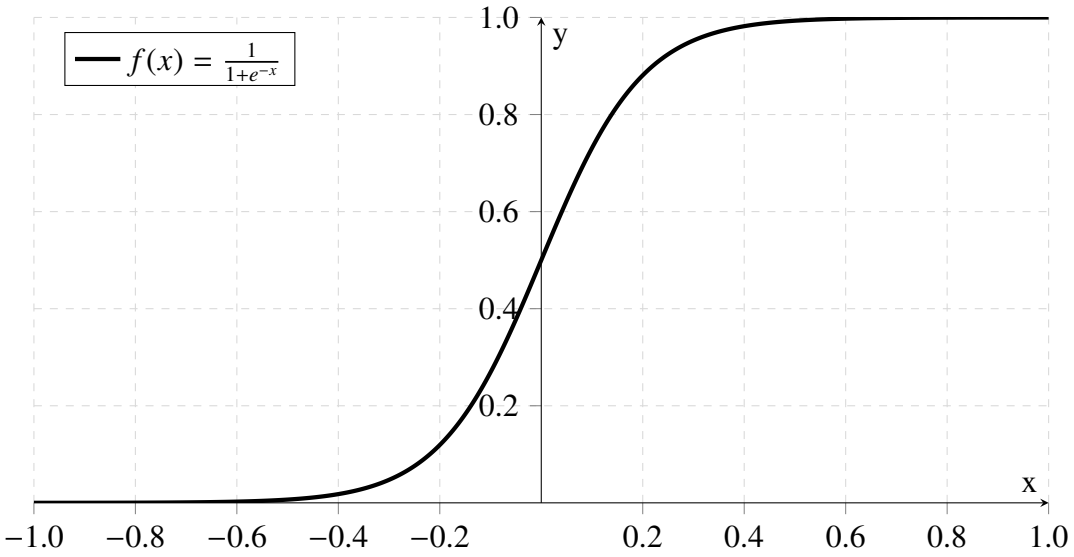

Figure 3.1: The sigmoid function used by the logistic regression algorithm.

Support Vector Machine

Support vector machine (SVM) is a classifier that was introduced around 25 years ago [29, 30]. It is widely used in many different fields such as speech recognition and information security. SVM's popularity is due to its ability to combat some problems in machine learning such as overfitting and the curse of dimensionality [31]. It is also well suited for complex datasets that are not too large [16].

The concept of an SVM is easiest explained in the linear case with a two-dimensional dataset. Such a dataset is visualised in Figure 3.2. This dataset contains two classes, one represented by the squares and one by the triangles. These two datasets are *linearly separable* since it is possible to draw a line, or *decision boundary*, completely separating the two classes [31]. Linear separability is the main requirement for the most basic form of SVM. In this case, the classifier seeks to find the decision boundary that maximises the margin between the two classes [30]. The margin is the distance between the two lines that are parallel to the decision boundary and pass through the sample(s) of each class that is closest to the decision boundary. In Figure 3.2 the margin is the distance between the dashed lines. The samples located on these dashed lines determine the properties of the decision boundary and are called *support vectors* [31], giving the algorithm its name.

The same method can be applied in higher dimensions, except that a hyperplane is used instead of a line. It is also possible to use a nonlinear function, either a polynomial or a similarity function such as the *radial basis function* (RBF), to separate the classes if the dataset requires a more complex model [31].

Naive Bayes

The naive Bayes model is a probabilistic classifier which has been studied for almost 70 years. It is widely used in retrieving information from text. This classifier is well suited for large datasets and can handle noise or missing data with no difficulties [15]. Furthermore,

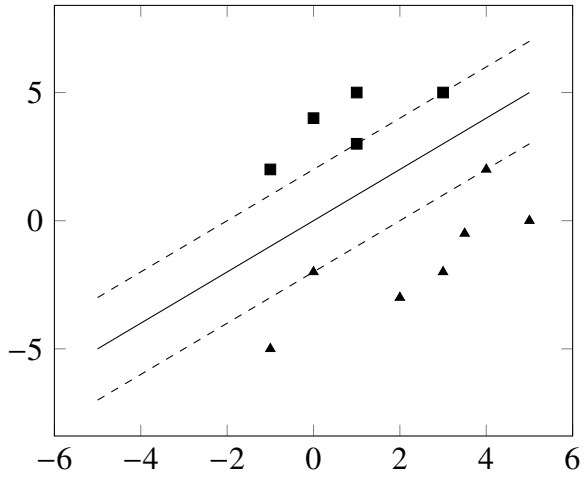

Figure 3.2: Separation of two classes using a support vector machine. The middle line is the decision boundary and the space between the two dashed lines, which pass through the support vectors, is the margin.

the naive Bayes classifier is fast and can compete with support vector machines when sufficient preprocessing of the data is performed [32].

This model chooses the class with the highest probability given the feature values as its prediction. The probability is calculated using Bayes' theorem

$$P(C_i|E) = \frac{P(C_i)P(E|C_i)}{P(E)} \tag{3.1}$$

where C_i is the class and E is the sample. In practice, it is useful to use this theorem because estimations of the probability for the three terms on the right-hand side are in many cases available [15]. Moreover, the naive Bayes classifier assumes that all features are independent thus meaning the theorem can be simplified to the following

$$P(C_i|E) = P(C_i)\prod_{j=1}^{a} P(A_j = v_{jk}|C_i) \tag{3.2}$$

where A_j is the feature and v_{jk} is the value.

In reality, the features are rarely independent of each other. But since the function in Equation 3.2 can minimise the misclassification error and due to the properties of the model in terms of simplicity and low resource usage, this is the classifier chosen in many situations [33]. The naive Bayes classifier can also outperform other models on smaller datasets but when the dataset grows the dominance of other algorithms, such as logistic regression, will start to show [34].

Decision Tree

A decision tree is a classifier which uses a tree structure and the first algorithm was introduced 55 years ago [35]. It is used in decision analysis and in machine learning as a way to find the best strategy for reaching a goal. In other words, it finds a decision based on the values of the feature vector [15].

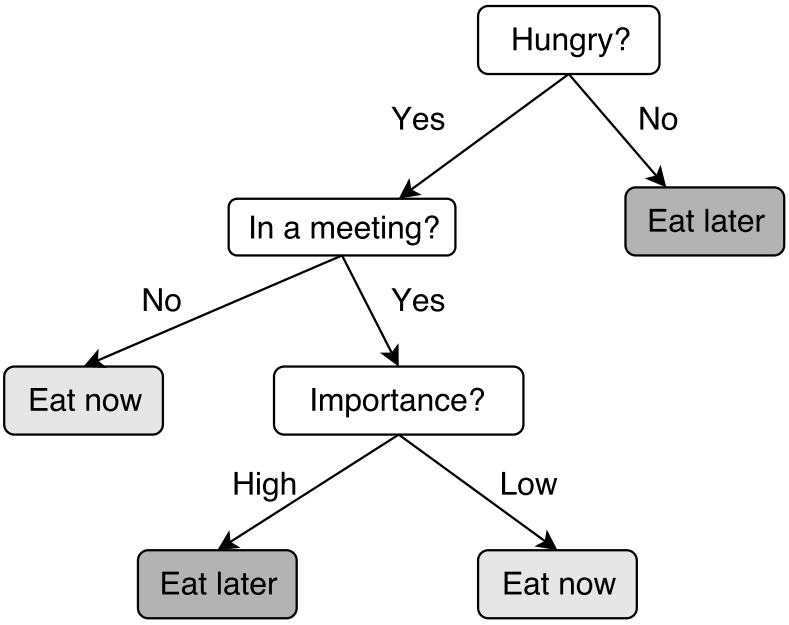

Figure 3.3: A decision tree for deciding whether to eat lunch now.

A simple Boolean decision tree is presented in Figure 3.3 for deciding whether to eat lunch now. There are two types of nodes in this tree: *answer nodes* and *decision nodes*. An answer node is a leaf node and represents a class whereas the decision node represents a feature test with branches to another decision tree for each value of this feature [36].

The goal of the algorithm is to find a decision tree that is both small and consistent. This is possible by using a greedy divide-and-conquer strategy which divides the problem into smaller subsets and tests the most important feature first. The most important feature is a feature which can make the most classifications in the sample. Decision trees are prone to overfitting when the dimensionality increases and the resulting trees are often complex [37]. To combat this problem, there exists a technique called *decision tree pruning* which eliminates irrelevant nodes [15].

Random Forest

The random forest classifier is an ensemble classifier that combines multiple classifiers into one [38]. As the name suggests, a random forest consists of a large number of decision tree classifiers. Each of the trees is trained on a randomly selected subset of the original dataset. During prediction, the most frequent result among the trees is chosen. The concept is similar to *bootstrap aggregating*, or *bagging* [39]. Bagging is a more general concept that can be used with any type of classifier, including decision trees. The random forest classifier differs from a bagging classifier using decision trees by introducing more randomisation during the training. Instead of selecting the most important feature out of all the features when deciding which feature to use in a decision node, the random forest algorithm only considers a random subset of features when making this selection. This makes the trees differ more from each other which generally leads to a better model. Random forests also make it possible to measure how important each of the features is to the result [16].

K-nearest Neighbours

The k-nearest neighbours classifier previously went by the name "minimum distance classifier" [40] or "proximity algorithm" [41]. For classification, the k nearest neighbours for a sample is calculated using a distance function [42]. The sample is then assigned the most frequent class among the neighbours. This is visualised in Figure 3.4 where the sample represented by the circle will be classified as a triangle since two of its three closest neighbours are triangles. Typically, k is chosen to be odd in order to avoid any risks of getting a tie [15]. The advantage of this classifier is that the training process is cheap, basically nonexistent, at the expense of being highly affected by the curse of dimensionality [43], as explained in Section 3.2.1.

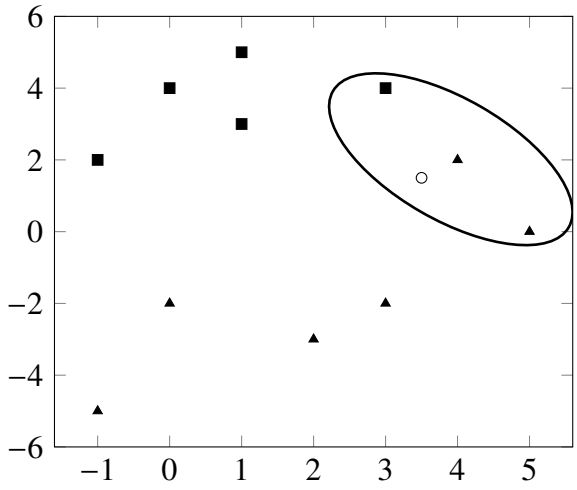

Figure 3.4: K-nearest neighbours for classification using two classes: rectangle and triangle. In this case of $k = 3$, the circle will be classified as a triangle.

3.3.2 Clustering

This section will present the purpose of clustering and the different types of algorithms utilised in this thesis. The purpose of cluster analysis is to divide the samples of a dataset into groups, or clusters. These clusters will contain samples that are similar to each other. Cluster analysis can aid the process of understanding, analysing and describing important information that may initially be hidden [44].

There exist multiple algorithms that utilise different strategies to compute clusters. These algorithms can be either *prototype-based*, *density-based* or *graph-based*. Prototype-based algorithms form clusters around prototypes which can be centroids (centres of clusters) or a probability distribution function. Density-based algorithms represent a cluster as a dense region of samples that are surrounded by regions with lower density. This is useful when the clusters are uneven in size or when there is noise in the data. Lastly, graph-based algorithms represent the data as a graph and the clusters as connected subgraphs [44]. The rest of the section will describe the evaluated clustering and visualisation algorithms.

K-Means

The K-means algorithm was developed by Lloyd [45] and originally intended for vector quantisation in signal processing [46] but is currently one of the most popular algorithms used for cluster analysis. The goal of the algorithm is to find k non-overlapping clusters.

The algorithm is prototype-based and uses centroids to define the clusters. A cluster centroid is computed as the mean of the data points, and data samples closest to the centroid are regarded as part of that cluster. The K-means algorithm works by first initialising k random centroids. Using a distance function, the data objects are then assigned to the closest centroid. The centroid is then updated by calculating the mean of the data samples in the new cluster. These steps are then repeated for a set number of iterations or until no data samples change clusters [44].

DBSCAN

Density-based spatial clustering of applications with noise (DBSCAN) is a density-based algorithm that was introduced more than 20 years ago [47]. An advantage compared to k-means is that there is no need to determine the number of clusters beforehand. DBSCAN can also detect clusters of different shapes [48]. These two qualities are helpful when there is not enough knowledge about the data.

When searching for clusters, DBSCAN defines the data samples as either *core samples*, *border samples* or *noise samples*. If an object has more than μ samples inside its ϵ-neighbourhood, for example a radius, it is considered as a core sample. If an object has less than μ sample inside its ϵ-neighbourhood and no core samples as neighbours it is considered as a noise sample. Otherwise, it is considered as a border sample [48].

The DBSCAN algorithm uses a seed list S which is a set of samples for expanding a cluster. In order to construct a cluster, DBSCAN first chooses a random unprocessed sample and inserts it into the empty S. A sample p is then removed from S and other samples that are inside the ϵ-neighbourhood of p and unprocessed are inserted into S, thus expanding the cluster. However, it is only the core samples that are used for expansion. A cluster is considered as complete when the seed list S is empty and a new search for another cluster begins until all samples are labelled [48].

Agglomerative Clustering

Agglomerative clustering is a method that creates a hierarchical structure between clusters and subclusters. It begins by placing each sample in its own cluster. The two most similar clusters are then selected and combined into a new cluster. The merging continues until one cluster remains. Along the way a hierarchical tree is constructed, with the root being the final cluster, the leaves being the single sample clusters and the forks along the way representing the merging of clusters [49]. The desired number of clusters can then be selected from this tree.

Visualisation

Another way of performing cluster analysis is to use algorithms that can visualise the similarity between samples in two or three dimensions. This can be useful if the results

of ordinary clustering algorithms are not satisfying. Finding the right parameters, such as the number of clusters, to use in a clustering algorithm is difficult. This often leads to less convincing clusters, and since the obtained results only specify which cluster a sample belongs to it can be difficult to draw conclusions about the quality of the clustering. Therefore, a visualisation algorithm can be used in conjunction with a clustering algorithm to provide a better understanding of how samples relate to each other [8].

Two algorithms that are commonly used for visualisation are *t-distributed stochastic neighbour embedding* (t-SNE) [50] and *multidimensional scaling* (MDS) [51]. They both seek to create a mapping from one space to another space with a lower dimension that preserves the similarity and dissimilarity between samples.

3.4 Evaluation

After a classifier has been trained or a clustering has been performed it is necessary to find a way of measuring its performance. This section will present the different methods and metrics used in our evaluation.

3.4.1 Classification

The most intuitive way of measuring the performance of a classifier is to make the classifier predict the results for each sample in the training set and measure the number of correct classifications. However, due to the model being optimised for the training data this approach will yield biased results that are not representative of how the model will perform on new data. Optimising a model for the training data is known as *overfitting* and generally leads to poor performance. To prevent overfitting, the model should be trained on a subset of the entire dataset while the rest is used for evaluation. A part of the dataset, usually 20%, is removed and saved as a *test set* that will not be seen by the model until the final evaluation [16]. This will reveal how well the model performs on data it has not seen before.

Another time at which evaluation is used is when comparing algorithms in order to determine which is best suited for a specific problem. Since the test set should not be used before the final evaluation it can not be used for this purpose. One alternative is to save a further part of the training set as a *validation set* and use it to compare different algorithms. However, there is a possibility that the remaining training set is too small to use for training unless there was an abundant amount of data to start with. A way to avoid reducing the training set is to use *cross-validation* instead of a validation set [16]. The most common form of cross-validation is *k-fold cross-validation*. In this process, the training set is first divided into k subsets. The classifier is then trained k times, each time excluding one of the subsets. The excluded subset is used as the validation set while the classifier is trained on the rest of the training set. The performance score is then the average of the scores for each of the k runs [52].

Metrics

The most basic metric that can be used to evaluate a classifier is the *accuracy*, which is the percentage of correct predictions. However, this number can be misleading for datasets containing classes of different frequency [16]. Therefore, other metrics, such as the *confusion matrix*, *precision*, *recall* and F_1*-score*, have to be considered as well. These metrics were originally designed to handle binary classifications where the predicted result for a sample can be interpreted as either positive or negative in regard to the problem. However, they can also be generalised to problems with more than two classes.

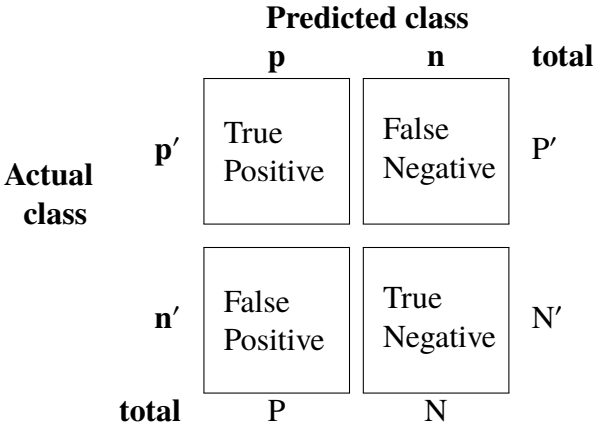

Figure 3.5: Binary confusion matrix.

A confusion matrix provides an overview of how good a classifier is at predicting results for the different classes and makes it easy to see if it often mixes up two or more classes. The rows in a confusion matrix represent the actual class of a sample and the columns represent the class that the sample was predicted as. In the binary case, this matrix contains the number of true positives, true negatives, false positives and false negatives with the structure presented in Figure 3.5. The number of true positives and true negatives are the numbers of correctly classified positive and negative samples. The number of false positives and false negatives are the numbers of positives wrongly classified as negatives and vice versa. These numbers are used in the calculations of a number of other metrics [53].

The confusion matrix can be used in the multiclass case by putting all classes on both axes. The matrix's diagonal will contain the number of correctly classified samples while the rest of the numbers show the cases where samples of one class are classified as belonging to another. This representation makes it possible to see if any classes are frequently mixed up.

Precision is a measurement of how good the classifier is at not wrongly classifying a negative sample as positive. In the binary case, it is calculated as:

$$Precision = \frac{TP}{TP + FP} \tag{3.3}$$

where TP is the number of true positives and FP is the number of false positives. The precision approaches one when the number of false positives approaches zero.

Recall is the complementary metric of precision. It measures how good the classifier is at finding all the sought after results. The recall is calculated as:

$$Recall = \frac{TP}{TP + FN} \tag{3.4}$$

The recall increases as the number of false negatives decreases, and a perfect score is obtained if all positives have been labelled as positive. It thus yields better scores for classifiers that find a large fraction of the positive samples, but does not penalise for incorrectly labelling samples as positive.

When precision and recall are used for multiclass classifiers the true and false positives and negatives are calculated for each class. There are two methods of combining these numbers into an average score: macro and micro. In macro-averaging, the precision and recall are computed for each class and the average score is the average of these scores. In micro-averaging, the sums of true and false positives and negatives over all classes are calculated and used in the formulae for precision and recall. Scores obtained by micro-averaging are biased towards the larger classes if there is a significant difference between the number of samples in each class. Macro-averaging treats all classes equally. The formulae are as follows:

$$Precision_{Macro} = \frac{\sum_{i=1}^{n} \frac{TP_i}{TP_i + FP_i}}{n} \tag{3.5}$$

$$Precision_{Micro} = \frac{\sum_{i=1}^{n} TP_i}{\sum_{i=1}^{n} (TP_i + FP_i)} \tag{3.6}$$

$$Recall_{Macro} = \frac{\sum_{i=1}^{n} \frac{TP_i}{TP_i + FN_i}}{n} \tag{3.7}$$

$$Recall_{Micro} = \frac{\sum_{i=1}^{n} TP_i}{\sum_{i=1}^{n} (TP_i + FN_i)} \tag{3.8}$$

where n is the number of classes.

The F_1 score is often used as a convenient way to compare classifiers. This metric is a combination of the precision and recall, by using the harmonic mean [54]. It is calculated as:

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{3.9}$$

3.4.2 Clustering

In this section, the evaluation methods and metrics for clustering will be presented. Evaluating the result of a clustering algorithm is not as straightforward as for classification. If the predicted and actual cluster would be compared for each sample the metrics of accuracy, precision and recall would yield poor results. Consider the vectors of cluster labels $[0, 0, 1, 1]$, $[1, 1, 0, 0]$ and $[2, 2, 3, 3]$. They all describe the same clustering, two clusters with the same two samples in each, but a pairwise evaluation would yield an F_1-score of 0. None of the labels match the one in their position in any of the other clusterings. Thus the evaluation metric described for classification are not applicable for clustering.

Instead, metrics that compare the quality of the clusters, and not the labelling of samples, are required.

Since there is no training and predicting involved in clustering there is no risk of overfitting on the data. This removes the need for a separate test set and cross-validation.

Metrics

One method of measuring the performance of clustering is to use the *adjusted Rand index* (ARI) which measures the similarity between two clusterings and ignores permutation. Let C denote the ground truth, K the cluster labelling, a the number of pairs of elements that are in the same cluster in C as in K, b the number of pairs of elements that are in different clusters in C and in different clusters in K and C_2^n the total number of pairs in the dataset. The Rand index (RI) is then computed according to the following formula [55]:

$$\mathrm{RI} = \frac{a + b}{C_2^n} \tag{3.10}$$

A disadvantage of using the Rand index is that there is no guarantee a random labelling will yield a score close to 0. To have this property an adjustment is made by discounting the expected Rand index, denoted as $E[\mathrm{RI}]$, as in the following formula [55]:

$$\mathrm{ARI} = \frac{\mathrm{RI} - E[\mathrm{RI}]}{\max(\mathrm{RI}) - E[\mathrm{RI}]} \tag{3.11}$$

Another way of measuring the similarity between two clusterings is to use the information theoretic concept of *mutual information* (MI). It is calculated as [56]:

$$\mathrm{MI}(U, V) = \sum_{i=1}^{|U|} \sum_{j=1}^{|V|} P(i, j) \log\left(\frac{P(i, j)}{P(i)P'(j)}\right) \tag{3.12}$$

where $P(i) = |U_i| \,/\, N$ and $P'(i) = |V_i| \,/\, N$ is the probability of a randomly chosen sample to fall into class i in the clusterings U and V respectively. $P(i, j) = |U_i \cap V_j| \,/\, N$ is the probability of a randomly chosen sample to fall into class i in U and class j in V.

The base version of mutual information suffers from the same disadvantages as the unadjusted RI. This is remedied in an identical way, resulting in the following formula for the *adjusted mutual information* (AMI) [56]:

$$\mathrm{AMI} = \frac{\mathrm{MI} - E[\mathrm{MI}]}{\max(H(U), H(V)) - E[\mathrm{MI}]} \tag{3.13}$$

where H is the entropy.

ARI and AMI yields scores between 0 and 1, and -1 and 1 respectively. In both cases a low score is poor and a score of one indicates a perfect match between the clusterings. Both ARI and AMI come with the disadvantage of requiring a correct clustering to compare with. In some cases, such as when determining the optimal number of clusters to use in the K-means algorithm on a completely new dataset, there is no correct solution to compare with. Those cases require a metric that can be computed without needing a

correct clustering. One such metric is the *silhouette score*. The silhouette score is a measurement of how dense and well separated clusters are [57]. It yields a value between -1 and 1 where a perfect clustering is represented by 1 and is calculated as [58]:

$$\frac{x - y}{max(x, y)} \tag{3.14}$$

where x is the mean distance between a sample and other samples in the same cluster and y is the mean distance between a sample and other samples in the nearest cluster.

Chapter 4
Data Preprocessing

The first step in our investigation was to examine the data produced by the verification process and transform it into a format that could be interpreted by machine learning algorithms. This chapter will present the data, explain its format and describe the process we used to transform it into numerical feature vectors. Dimensionality reduction of the feature vector will also be described.

4.1 Data

When a test case is executed it produces artefacts that contain details about the test. Examples of these artefacts are log files and waveforms. In order to obtain as much information as possible all artefacts would need to be examined, but to limit the scope of this thesis only log files were used. Log files provide a good overview of test executions and only very specific information is lost by only using them.

```
Error-[FCIBH] Illegal bin hit
/h/harryd/gitrep/panda_tfa_dev/src/vip/ipp/phase_out/tb/top/ax_tb_ph_cov.sv, 194
ax_tb_ph_cov, "ax_tb_ph_cov::cov_collector::cg_cfg"
  VERIFICATION ERROR (FUNCTIONAL COVERAGE) : At time 13203959 ps, Illegal
  state bin ig of coverpoint cp_nr_exp in covergroup
  ax_tb_ph_cov::cov_collector::cg_cfg got hit with value 0x5
  Covergroup Instance: cfg
  Design hierarchy: ax_tb_ph_cov
```

Figure 4.1: Example message from a log file

The log files produced by the test benches we investigated consist of log messages that typically specify a severity level, a file path, a line number, a timestamp, a tag and a descriptive text. Examples of the severity levels are *fatal*, *error*, *warning* and *info*. The file path, line number and timestamp indicate where and when the message was produced. The tag describes the type of the message. At the end of the log message is a descriptive

text that provides further information. An example message from a log file can be seen in Figure 4.1.

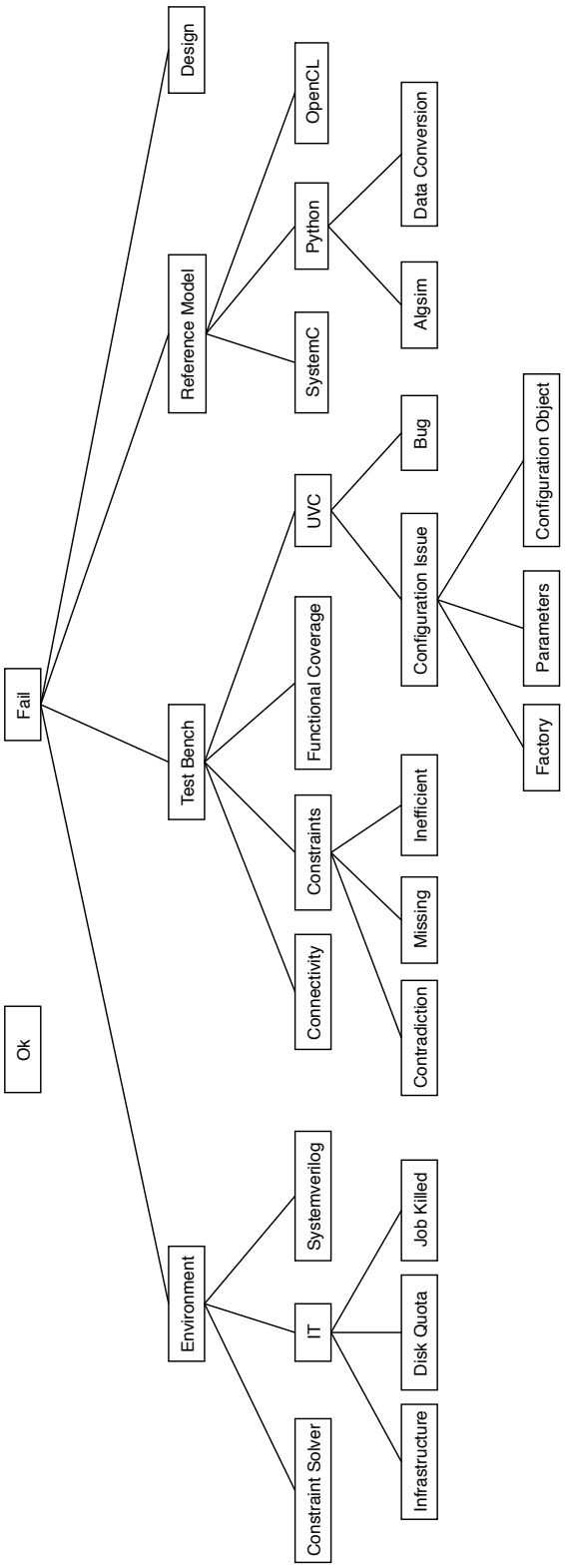

Figure 4.2: Taxonomy of the root cause for test case failures

The root causes of test failures are not completely separate from each other, but can be grouped together into larger categories that represent the part of the system they appear in. This enabled a construction of a tree whose levels can be used for classifications of different granularity. The tree, along with the root cause labels used can be found in Figure 4.2. The tree is a representation of the most common root causes the verification engineers at Axis could identify. At the lowest granularity level, test cases are only classified as passing or failing. When the granularity level is increased, the failures are divided into four classes. Each of the four classes indicate the location of the root cause. Three of the four classes can then be further divided into classes that provide a more detailed description of where the root cause is located. It would be desirable to classify root causes as one of the leaf node labels, the highest granularity level, but this may not always be possible. In cases where the algorithm is unable to distinguish between two or more leaves it could be possible to use the parent node label, which represents the part of the system the leaves are located in. Thus some information about where the flaw is located would still be provided.

When the root cause taxonomy tree had been created it was possible to begin generating the test cases for our dataset. The test cases were generated by injecting one type of flaw at a time into working test benches or designs. The injected flaws were similar to flaws that had been encountered by the verification engineers in the past. This resulted in a set of test cases where each failure had a single known root cause and could be labelled accordingly. The labels were constructed by finding the path between the root node and the leaf representing the root cause in the tree from Figure 4.2 and concatenating the labels on the nodes. For example, failures caused by a flaw in the OpenCL part of the reference model received the label `Fail:Reference_Model:OpenCL` and the ones caused by an incorrectly implemented test bench component were labelled with `Fail:Test_Bench:UVC:Bug`. Design flaws were labelled as `Fail:Test_Bench:RTL:Bug`. The construction of the taxonomy tree and the generation of our dataset were performed by the verification engineers at Axis. Due to time constraints, generating test cases for all leaves in the taxonomy tree was not possible. The labels for the subset of root causes used in our evaluation and the number of samples per label in the dataset are presented in Table 4.1.

Table 4.1: The labels for the test cases used in the evaluation and the number of samples per label in the dataset.

Label	Number of samples
Fail:Reference_Model:Python:Data_Conversion	1063
Fail:Reference_Model:OpenCL	620
Fail:Reference_Model:SystemC	695
Fail:Test_Bench:Functional_Coverage	1129
Fail:Test_Bench:Constraints:Contradiction	1367
Fail:Test_Bench:Constraints:Missing	1472
Fail:Test_Bench:UVC:Bug	1203
Fail:Test_Bench:UVC:Configuration_Issue:Configuration_Object	897
Fail:Test_Bench:UVC:Configuration_Issue:Parameters	456
Fail:Test_Bench:UVC:Configuration_Issue:Factory	963
Fail:Test_Bench:RTL:Bug	1405
OK	1256

The test cases were generated by a set of different test benches that tested different image processing designs. These test benches were selected by the verification engineers at Axis because they were structurally similar and thus the log files generated by them would also be similar.

4.2 Transformation

In this section, the transformation of text in log files to numerical values is described. In order to be interpreted by machine learning algorithms, the input data needs to be represented by numerical feature vectors. Therefore, each log file had to be transformed into such a vector. The process of transforming the contents of a log file into a vector of numerical features is called *abstraction*. The first step of the abstraction process is to create a set of regular expressions to match frequently appearing patterns. Log messages consist of two parts: a static message type and a variable parameter part [59]. The regular expressions are created to match the static parts of the messages while allowing different values for the parameters. Creation of the regular expressions can be performed either by using an automated approach, where a tool tries to extract the expressions or by using system knowledge in order to manually create the expressions [60].

In our investigation, trials with the *LogCluster* [61] tool were performed but did not produce any satisfying results. The log files we used contained several textual representations of large arrays and data structures in which the tool recognised patterns that were not helpful. Therefore, we created our own regular expressions with the help of the verification engineers at Axis.

The second step in the abstraction process is to use the created regular expressions to extract specific parts which can be viewed as the representative for a message. Thus, messages that differ by one or two variable values can have the same representative and be treated as belonging to the same message category, reducing the number of categories. For example, the expression we created to match messages of the type shown in Figure 4.1 was the following:

```
([^-]+)-\[([^\]]+)\]
```
(4.1)

As seen in Figure 4.3, the expression extracted the message severity, tag, file path and line number. The extracted information was concatenated and the resulting string was seen as the representative of the message category. In some cases, the timestamp was also extracted to determine if the message was written during the set-up phase or during the actual execution.

After the abstraction process was completed the frequency of every category was determined and added as a feature which can be observed in Figure 4.3. Counting the frequency of words or phrases is often used in document classification in the natural language processing field [19]. The configuration parameters used when running the test case were also extracted and added as features, along with Boolean values indicating if the test case failed, crashed or produced a stacktrace.

The final step of the transformation process was scaling the data, now in the form of numerical feature vectors, to be centred around the mean and have unit variance. The scaling was performed using the `StandardScaler` module from the machine learning library

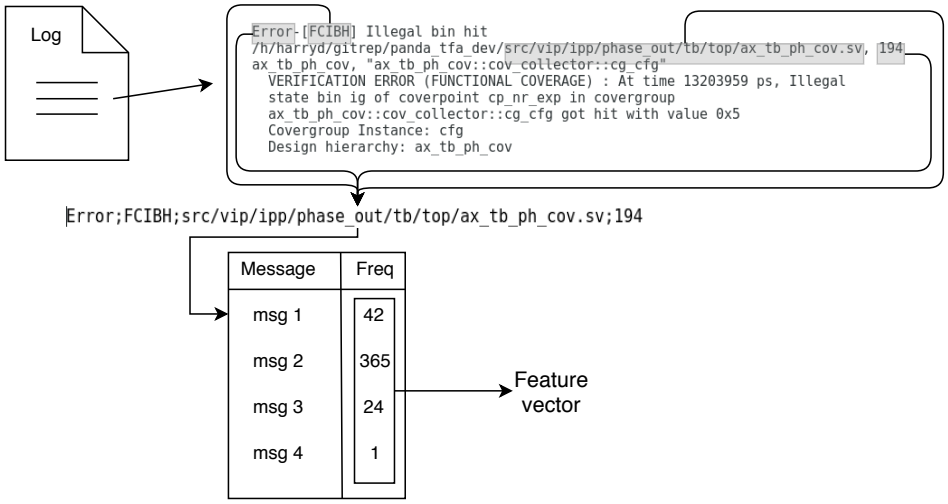

Figure 4.3: Message extraction and transformation from log file to a numerical vector.

in Python called Scikit-learn [62]. Some algorithms, such as support vector machines, perform worse if the axes of the dimensions have very different scales [16].

The transformation process resulted in a large feature set containing 616 different features. To reduce the number of features we performed an initial dimensionality reduction. With the help of engineers with domain knowledge, log messages with a low severity level had the line number removed and resulted in a feature vector of 287 dimensions. An investigation using a small subset of the data indicated promising results compared to using the full feature set. The feature vector of 287 dimensions was chosen as our baseline for the feature set.

Chapter 5
Evaluation

This chapter will describe our evaluation process and present the results. First, the evaluation process and results for the classification algorithms will be presented. Then the evaluation process and results for the clustering algorithms will be presented. Finally, a description of how the best clustering and classification algorithms were used in the implementation of our tool for automatic clustering and classification of test failures will be provided. The evaluation was performed using a dataset consisting of 12500 test cases labelled with their root causes. The evaluation and our tool implementation were performed using Python and Scikit-learn. Implementations of all machine learning algorithms were available in Scikit learn and these implementations were used in our evaluation and tool implementation.

5.1 Classification

This section will describe the three steps in our evaluation of the classification algorithms. First we evaluated eight algorithms in order to determine the three most suitable for our problem. These three algorithms were then optimised to improve their performance. The optimised algorithms were then used in a final evaluation. Before the evaluation process, the dataset was divided into a training set containing 10000 samples and a test set containing 2500 samples. The training set was used with cross-validation in the first two evaluation steps and the test set was used in the final evaluation.

5.1.1 Algorithm Selection

As the first step in the process of evaluating the classification algorithms we evaluated and compared the algorithms in Table 5.1. The purpose of this evaluation was to get an overview of how well the different algorithms performed for our problem. Based on the

results, we wanted to select the three algorithms that appeared to be best suited for the problem and proceed with them to the next step.

The evaluated classification algorithms, along with the names of their implementing functions in the Scikit-learn library, are presented in Table 5.1. These algorithms were selected because they all utilise different strategies that may perform differently on different types of data. To modify their behaviour, the algorithms use adjustable hyperparameters. For this part of the evaluation, we used the algorithms with the default hyperparameter values provided by Scikit-learn.

Table 5.1: The evaluated classification algorithms and the names of their corresponding Scikit-learn functions.

Algorithm	Scikit-learn function
Random forest	RandomForestClassifier
Polynomial suport vector machine	SVC(kernel='poly')
RBF support vector machine	SVC(kernel='rbf')
Linear support vector machine	LinearSVC
Decision tree	DecisionTreeClassifier
Logistic regression	LogisticRegression
K-nearest neighbours	KNeighborsClassifier
Naive Bayes	GaussianNB

Along with the classification algorithms, we evaluated three dimensionality reduction algorithms in order to determine their impact on classification performance and computation time. The evaluated dimensionality reduction algorithms are presented in Table 5.2. We decided to use algorithms utilising certain thresholds to select the most important features. The feature selection algorithms we evaluated were `PCA` and `SelectFromModel`. `PCA`, as described in Section 3.2.1, uses a mathematical method to reduce the dimensionality of the feature set. `SelectFromModel` is a wrapper method implemented in the Scikit-learn library. It selects features by training a model using all the features and then selecting the features that were the most important for the model. We used this method with two models, one random forest and one support vector machine model.

Table 5.2: The evaluated feature selection algorithms along with their corresponding Scikit-learn functions and the abbreviations used in the result table.

Algorithm	Scikit-learn function	Abbreviation
Principle component analysis	PCA	PCA
Most relevant features, random forest	SelectFromModel()	SFM RF
Most relevant features, linear SVM	SelectFromModel()	SFM SVC

Results

The results for the classification algorithms are presented in Table 5.3. Each algorithm was first evaluated using the baseline feature set, and then once using each of the dimensionality reduction methods from Table 5.2. In Table 5.3, the algorithms are grouped by the feature set used in the evaluation. For each algorithm, the accuracy, precision, recall, F_1-score, training time and prediction time were calculated using 5-fold cross-validation and are presented in Table 5.3.

In Table 5.3 it can be observed that the three algorithms that yielded the highest F_1-score for the classification metrics were random forest, decision tree and K-nearest neighbours. Almost every algorithm had the score for the classification metrics negatively impacted by dimensionality reduction. However, dimensionality reduction reduced training and prediction times for all algorithms except the random forest and decision tree algorithms, which suffered longer training times when used with PCA. For example, the training time for the linear SVM was reduced from 72 to 20 seconds and the prediction time of the k-nearest neighbour algorithm was reduced from 56 to 1.7 seconds. The algorithms with the lowest training and prediction times were the naive Bayes and the decision tree.

5.1.2 Optimisation

In this section, the optimisation of the classification algorithms will be presented. Since the initial evaluation was performed with the default hyperparameters for all algorithms we improved the performance of the three best algorithms from the initial evaluation by finding the optimal hyperparameters. The classification algorithms with the highest F_1-score in the initial evaluation were random forest, decision tree and k-nearest neighbours. The k-nearest neighbours classifier yielded slightly lower scores when PCA was applied, but the computation time was significantly lower. Therefore, we chose to use k-nearest neighbours with PCA for the optimisation. We chose the three algorithms with the highest F_1-score since there was a difference between the top three and the other algorithms. As seen in Figure 5.3, the naive Bayes algorithm had F_1-score of 0.6, the SVM and the logistic regression had longer computation time compared to the top three algorithms. The results for the non-optimised versions of the three algorithms are in bold in Table 5.3.

The decision tree and k-nearest neighbours algorithms had small parameter spaces thus an exhaustive search of all parameter combinations was feasible. For the random forest algorithm, which had a larger parameter space, we decided to use a random search. Random search is an effective method for optimising hyperparameters in large parameter spaces [63]. Both search methods work by selecting combinations, either randomly or exhaustively, of parameters from a specified parameter space and evaluating the performance of the classifier for those combinations.

Table 5.3: Results for the classification algorithms, grouped by the feature set used in the evaluation. The three that had the highest F_1-score and were chosen for optimisation are marked in bold.

Classifier	Accuracy	Precision	Recall	F_1-score	Train (s)	Predict (s)
Baseline feature set						
Random forest	**0.899**	**0.907**	**0.904**	**0.905**	**0.277**	**0.132**
SVC poly	0.556	0.864	0.560	0.609	52.655	56.315
SVC rbf	0.806	0.845	0.800	0.813	17.311	36.422
LinearSVC	0.851	0.856	0.852	0.852	72.463	0.184
Decision tree	**0.892**	**0.901**	**0.899**	**0.899**	**0.342**	**0.067**
Logistic regression	0.841	0.851	0.840	0.842	62.498	0.191
K-neighbours	0.883	0.890	0.887	0.888	0.522	56.618
Naive Bayes	0.643	0.763	0.652	0.607	0.180	0.933
Dimensionality reduction using PCA						
Random forest	0.891	0.899	0.896	0.897	0.513	0.100
SVC poly	0.760	0.851	0.756	0.779	9.809	15.468
SVC rbf	0.808	0.844	0.803	0.814	5.971	12.239
LinearSVC	0.779	0.801	0.768	0.778	37.075	0.189
Decision tree	0.885	0.893	0.891	0.892	0.806	0.058
Logistic regression	0.794	0.817	0.785	0.793	31.105	0.127
K-neighbours	**0.882**	**0.890**	**0.885**	**0.887**	**0.111**	**1.683**
Naive Bayes	0.518	0.725	0.510	0.514	0.082	0.338
Dimensionality reduction using SFM SVC						
Random forest	0.893	0.901	0.899	0.900	0.133	0.096
SVC poly	0.650	0.810	0.643	0.675	7.732	12.853
SVC rbf	0.747	0.791	0.752	0.757	4.520	10.773
LinearSVC	0.786	0.790	0.791	0.786	26.048	0.156
Decision tree	0.887	0.897	0.895	0.896	0.121	0.057
Logistic regression	0.752	0.775	0.757	0.757	21.339	0.195
K-neighbours	0.877	0.882	0.883	0.882	0.270	16.335
Naive Bayes	0.536	0.608	0.579	0.509	0.059	0.234
Dimensionality reduction using SFM RF						
Random forest	0.896	0.904	0.902	0.902	0.126	0.101
SVC poly	0.704	0.820	0.688	0.719	3.813	6.410
SVC rbf	0.781	0.820	0.776	0.788	2.912	6.284
LinearSVC	0.786	0.798	0.784	0.786	19.937	0.136
Decision tree	0.891	0.899	0.897	0.898	0.099	0.058
Logistic regression	0.772	0.786	0.766	0.769	15.400	0.181
K-neighbours	0.878	0.884	0.883	0.883	0.112	4.668
Naive Bayes	0.575	0.677	0.610	0.534	0.044	0.192

Results

The results after the optimisation of the three best classification algorithms are presented in Table 5.4. The method and metrics used for evaluation were the same as in the previous section, i.e. 5-fold cross-validation and the metrics accuracy, precision, recall, F_1-score, training time and prediction time. In Table 5.4, the relative changes for each metric compared to the non-optimised version of the algorithm are presented.

Table 5.4: Results and relative change after optimisation of the random forest, decision tree and K-neighbours classifiers.

Classifier	Accuracy	Precision	Recall	F_1-score	Train (s)	Predict (s)
Random forest	0.907 (+0.9%)	0.915 (+0.9%)	0.913 (+1.0%)	0.913 (+0.9%)	8.410 (+2936.1%)	2.074 (+1471.2%)
Decision Tree	0.896 (+0.4%)	0.904 (+0.3%)	0.902 (+0.3%)	0.902 (+0.3%)	0.385 (+12.6%)	0.113 (+68.7%)
K-neighbours	0.885 (+0.3%)	0.891 (+0.1%)	0.891 (+0.7%)	0.891 (+0.5%)	0.101 (-9.0%)	0.994 (-41.0%)

As can be observed in Table 5.4, optimising the algorithms slightly increased their scores for the classification metrics. Optimisation increased the computation time for the random forest and the decision tree classifier whereas it decreased for the k-neighbours classifier. The random forest classifier responded best to optimisation with regard to the classification metrics.

5.1.3 Final Evaluation

The final evaluation was performed by training the algorithms on the entire training set and predicting labels for the test set. The results are presented in Table 5.5.

Table 5.5: Results for the final evaluation of the random forest, decision tree and K-neighbours classifiers.

Classifier	Accuracy	Precision	Recall	F_1-score	Train (s)	Predict (s)
Random forest	0.907	0.916	0.912	0.913	5.344	0.182
Decision Tree	0.895	0.902	0.900	0.900	0.111	0.002
K-neighbours	0.880	0.888	0.886	0.886	0.059	0.113

The results of the final evaluation indicated that the random forest algorithm is the best classifier for root cause classification. As observed in Tables 5.4 and 5.5, the results of the final evaluation were only marginally lower than the results obtained by using cross-validation on the training set, indicating that all three classifiers generalise well to new data.

When the random forest classifier had been established as the best classifier for this problem, further investigations of its performance were made. The next paragraphs will present a confusion matrix, the importance of the number of samples in a dataset, the dependency between the number of trees and the computation time. The performance of the random forest classifier at different granularity levels will also be presented.

A confusion matrix for the random forest classifier is presented in Figure 5.1. The confusion matrix illustrates how well the classifier performed for the different root causes. The predicted labels are on the x-axis and the true labels are on the y-axis. Data conversion, OpenCL and register transfer level (RTL) bugs were the most difficult root causes to classify. Data conversion was often mistaken for either an RTL bug, which is a bug in the design, or a universal verification component (UVC) bug, which is a bug in the test bench. Contradicting constraints in the test bench could be misclassified as missing constraints.

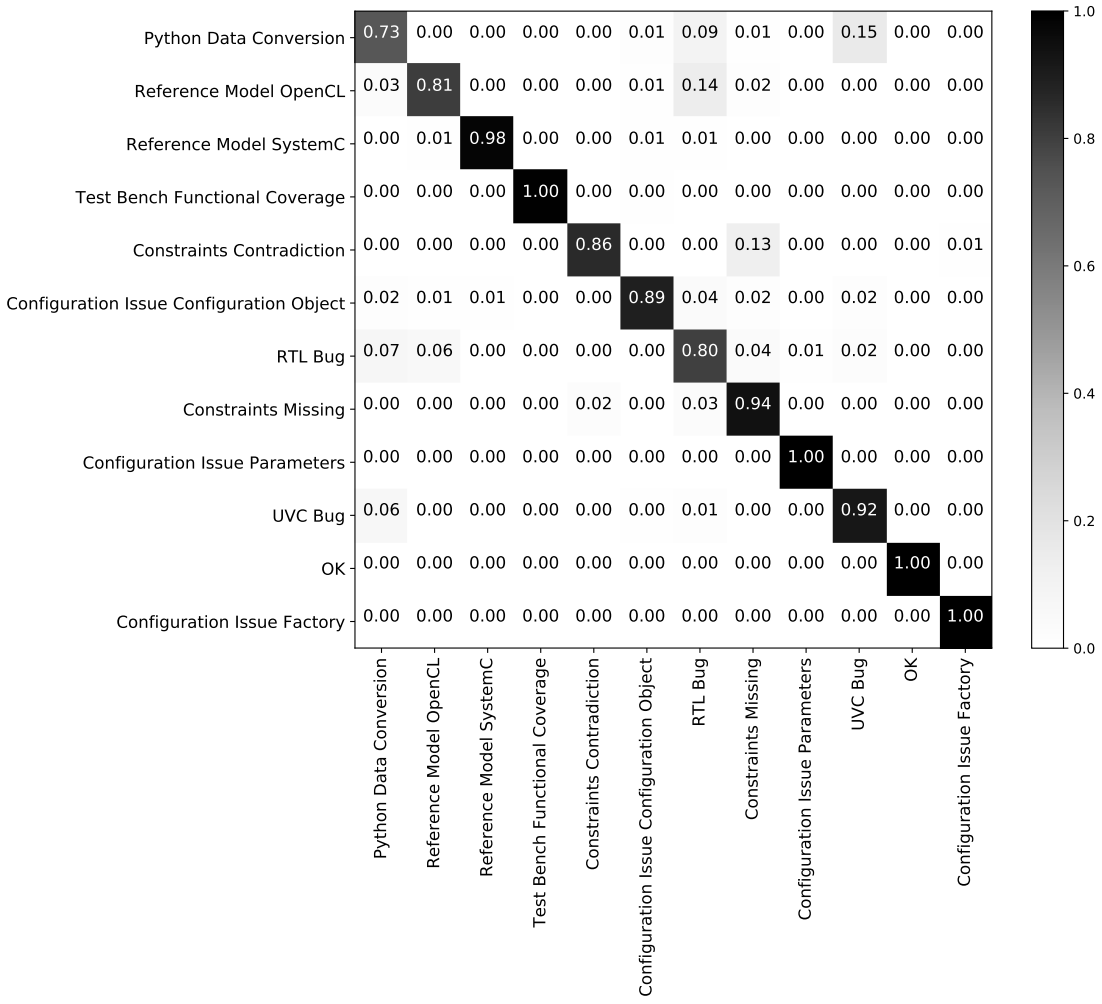

Figure 5.1: Confusion matrix for the random forest classifier.

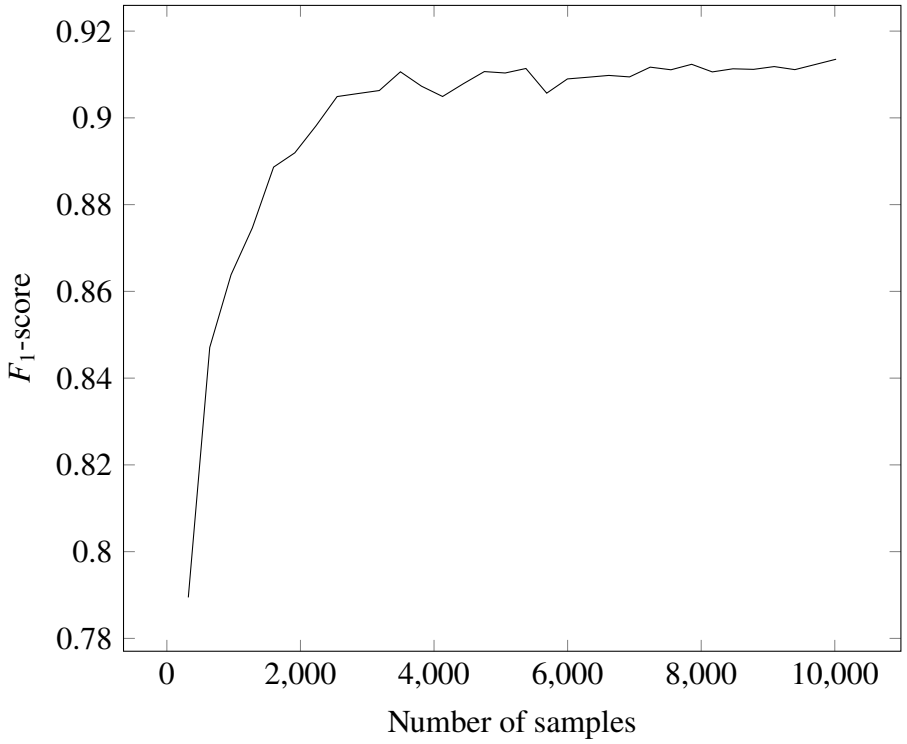

Figure 5.2: Illustration of how F_1-score scales with the number of data samples when using a random forest classifier.

The significance of data is illustrated in Figure 5.2. When the dataset reached 3000 samples the performance gained by adding more samples diminished. The F_1-score fluctuates when the number of samples is between 3000 and 6000 but stabilises after that.

As observed in Table 5.4, the time required for both training and predicting increased for the random forest algorithm after it was optimised. The reason for this increase was that the value of the hyperparameter determining the number of trees in the forest changed from 10 to 300 during optimisation. The significance of the number of trees in a random forest was further investigated by measuring the computation time, which is the time required for both training and prediction, and F_1 score for different numbers of trees. The results are illustrated in Figure 5.3. There it can be seen that the computation time increased linearly with the number of trees while he F_1-score remained roughly the same.

The results of the random forest classifier using the different granularity levels from the taxonomy tree in Figure 4.2 are presented in Table 5.6. Classification at lower granularity levels yields better accuracy scores but lower F_1-scores. A granularity level of one, classification of passing and failing test cases, results in a perfect score. However, it should be noted that whether a test failed or not is written in the log file. Thus the lowest granularity level, while perfect, does not aid debugging.

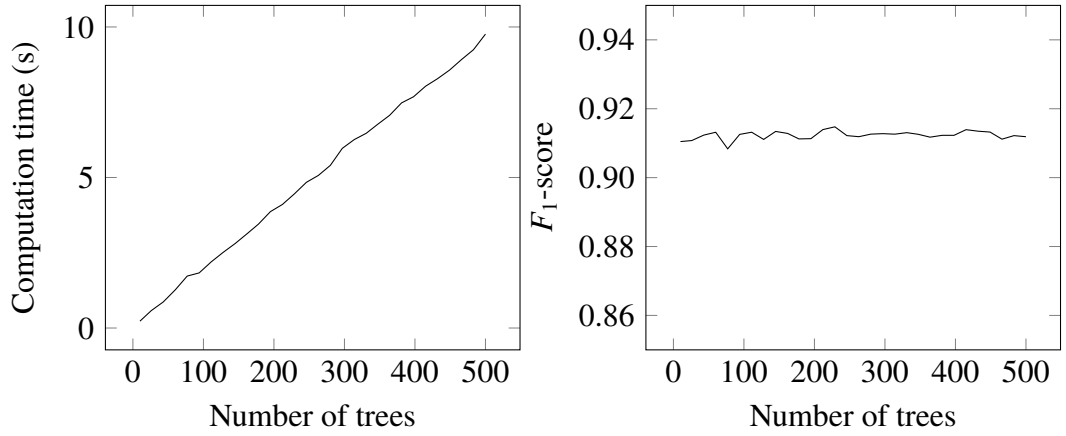

Figure 5.3: Illustration of how computation time and F_1-score scales with the number of decision trees in a random forest classifier.

Table 5.6: Results of evaluation using different granularity levels in the class taxonomy tree with the random forest classifier.

Granularity level	Accuracy	Precision	Recall	F_1-score
5	0.907	0.916	0.912	0.913
4	0.904	0.902	0.898	0.900
3	0.923	0.913	0.903	0.908
2	0.928	0.909	0.903	0.908
1	1.00	1.00	1.00	1.00

5.2 Clustering

In this section, the evaluation process and results for the clustering algorithms will be presented. Evaluating the clustering algorithms only required one step since there is no training process involved in clustering, and thus no risk of overfitting. Therefore, one evaluation using the whole dataset could be performed. The evaluated clustering algorithms are presented in Table 5.7.

Table 5.7: The evaluated clustering algorithms and the names of the corresponding Scikit-learn functions.

Algorithm	Scikit-learn function
K-means	KMeans
DBSCAN	DBSCAN
Agglomerative clustering	AgglomerativeClustering

The K-means and agglomerative clustering algorithms require the hyperparameter determining the number of clusters to be specified. To determine the optimal number of

clusters the silhouette score was utilised. Clusterings were computed with different values for the hyperparameter, and the value yielding the highest silhouette score was chosen as the optimum. To avoid bias towards the K-means and the agglomerative clustering algorithms, the same method was applied for the μ and ϵ hyperparameters of DBSCAN.

When optimal hyperparameter values for the algorithms had been obtained, the clusterings computed by all algorithms were compared to the correct clustering, the *ground truth*, using the AMI and ARI metrics. Since the goal of clustering was to cluster tests that failed due to the same actual root cause, as opposed to the classification goal of determining the type of root cause, it was not possible to only use the same labels used for classification as the ground truth. Instead, we utilised that our data was generated by injecting one root cause at a time into working designs or test benches. Thus we could consider all failing tests created together as a correct cluster.

The only dimensionality reduction method evaluated for clustering was PCA. The `SelectFromModel()`-functions work by training and using a classifier to determine the most important features and were therefore not considered for clustering.

5.2.1 Results

Each of the clustering algorithms was first evaluated using the baseline feature set and then evaluated when PCA was used for dimensionality reduction. Each algorithm was evaluated using the AMI score, the ARI score and the computation time. The results of both evaluations are presented in Table 5.8.

Table 5.8: Results for the clustering algorithms. The best performing algorithm along with the best results for each metric are written in bold.

Algorithm	AMI	ARI	Computation time (s)
Baseline feature set			
K-means	0.505	0.480	0.079
DBSCAN	0.568	0.530	0.086
Agglomerative Clustering	0.540	0.515	0.036
Dimensionality reduction using PCA			
K-means	0.543	0.513	0.041
DBSCAN	**0.593**	**0.545**	0.007
Agglomerative Clustering	0.543	0.519	**0.006**

As can be observed in Table 5.8, all algorithms performed better when PCA was used for dimensionality reduction. Both DBSCAN and the agglomerative clustering had shorter computation times compared to the K-means algorithm when PCA was applied. DBSCAN in conjunction with PCA yielded the highest AMI and ARI scores and was only one millisecond slower than the agglomerative clustering, which indicated that it was the best clustering algorithm for this problem.

5.3 Tool Implementation

The most suitable clustering algorithm, DBSCAN, and the most suitable classification algorithm, random forest, were used to implement a tool for automatic clustering and classification of root causes. The tool is able to transform textual data into a numerical feature vector. The numerical feature vector is applied in clustering and classification of the test failures. The rest of the section will describe how clustering was used in conjunction with visualisation and how classification was implemented with a confidence level.

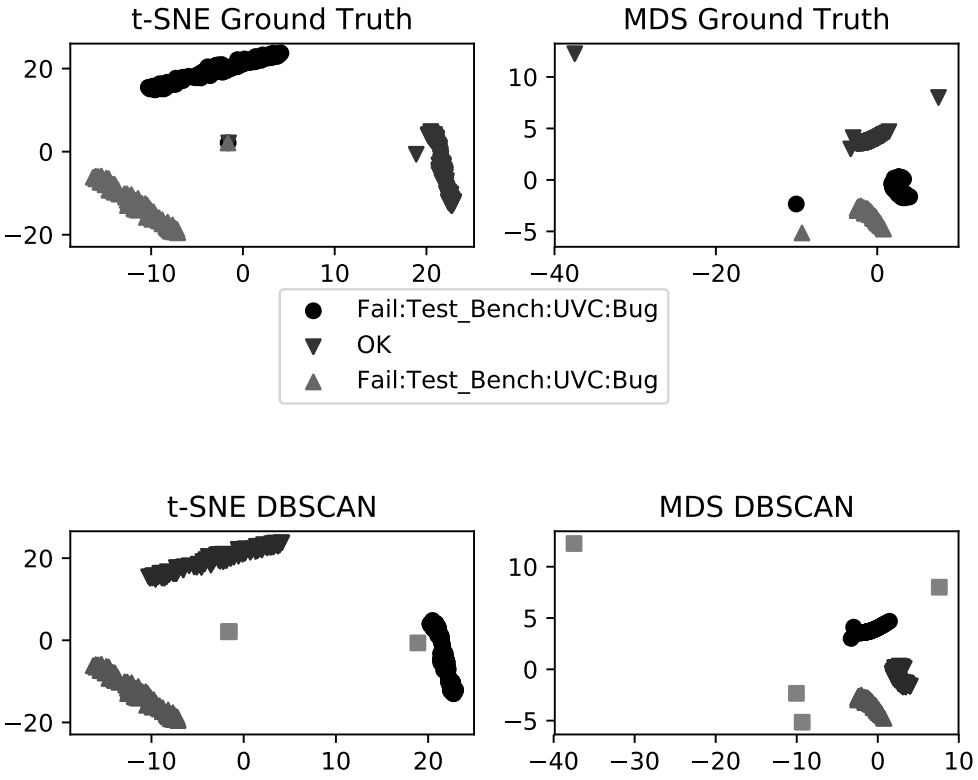

Figure 5.4: Clustering of test failures in combination with MDS and t-SNE. The clustering received an AMI score of 0.940 and an ARI score of 0.980.

Clustering of test failures is combined with the visualisation algorithms MDS and t-SNE. An example can be seen in Figure 5.4, where the clustering computed by DBSCAN is compared to the ground truth. The legend describes the root causes for the ground truth. There is no legend attached to the bottom figures which visualise the clusters DBSCAN found. There is one set of passing test cases as well as two sets of test cases that fail due to two UVC bugs. A UVC bug refers to a flaw in the test bench components, as mentioned in Section 2.3. The clustering produced by DBSCAN is almost perfect and yielded an AMI score of 0.940 and an ARI score of 0.980. As can be observed in the images generated by the visualisation algorithms, the samples that DBSCAN identifies as a fourth cluster, the

single dots, appear to differ from the clusters they belong to. This observation would not have been possible without the visualisation algorithms, indicating their usefulness.

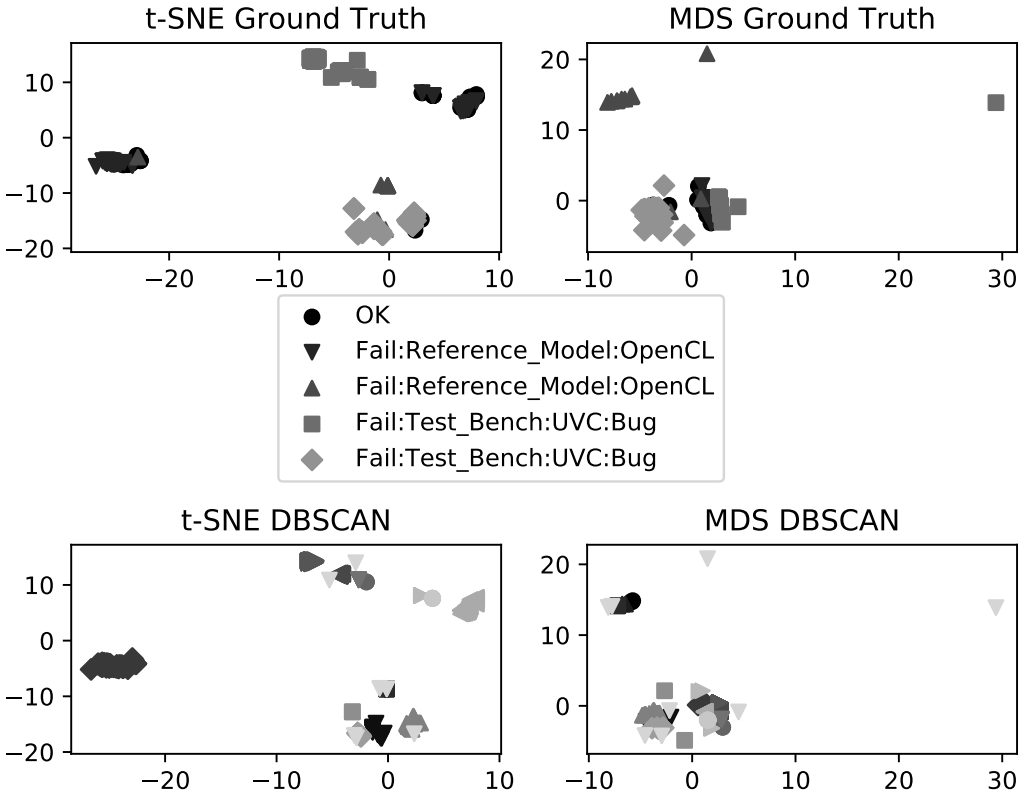

Figure 5.5: Clustering of test failures in combination with MDS and t-SNE. The clustering received an AMI score of 0.422 and an ARI score of 0.366.

Figure 5.4 does however show an unusually good clustering. The average AMI and ARI scores obtained by the evaluation were 0.593 and 0.545. Thus, many clusterings will be similar to the one in Figure 5.5, which received an AMI score of 0.422 and an ARI score of 0.366. Figure 5.5 has the same layout as Figure 5.4. For the samples in Figure 5.5, neither the clustering algorithm nor the visualisation algorithm produce satisfying results. In the two upper plots, which contain visualisations of the ground truth, it can be seen that many clusters overlap in the visualisation. The two bottom plots, which show the clusters computed by the DBSCAN algorithm, reveal that the clusters computed by DBSCAN differ from the ground truth. The two cases presented in Figures 5.4 and 5.5 show that the performance of both DBSCAN and the visualisation algorithms varies depending on the samples used.

When predicting results, the random forest classifier is able to estimate its confidence about each prediction. In Figure 5.6 the relationship between confidence and misclassification rate is illustrated. The misclassification rates were obtained by using the random forest algorithm trained on the training set to predict outputs for the test set. The rates for

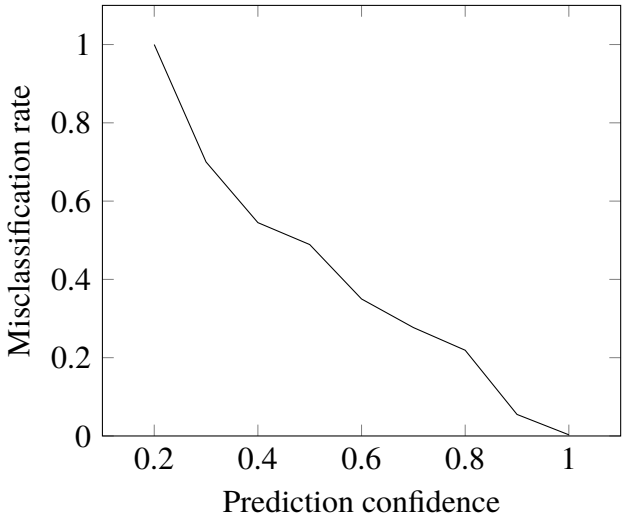

Figure 5.6: Illustration of the dependence between prediction confidence and misclassification rate.

each confidence level were then calculated by dividing the number of misclassifications made at that confidence level by the total number of classifications at that level. These results suggest that the confidence can be used to indicate how well a prediction can be trusted. The confidence level was therefore used to order the predictions and create a priority list for debugging.

Chapter 6

Discussion

To the best of our knowledge, the application of machine learning techniques to clustering and classification of test failures in hardware verification has not been investigated before. Therefore an exact comparison with prior work is not possible.

6.1 Data

The dataset was generated by injecting flaws into different image processing designs and test benches. The used test benches were structurally similar and generated log files with the same message types. We did not examine every detail of these log files, so it might be possible that log files created by some test benches contained details not present in log files from other test benches. This could have led to the classification algorithms associating specific test benches with a root cause, which should not be the case and could have biased the results. The usage of different image processing designs when generating the dataset could also have affected our results since the classifier could have associated a design with a root cause.

Both the aforementioned effects could have been avoided by using the same design and test bench to generate all test cases. However, the machine learning algorithms could learn root causes only specific to this design and test bench thus having difficulties classifying or clustering test failures from other designs. By using many different designs and test benches we wanted to obtain a more general result. Instead of indicating how well machine learning can be applied to root cause analysis of failures from a specific test bench, our results indicate the performance for test benches with a certain structure.

6.2 Classification

The random forest classifier yielded accuracy and F_1 scores over 0.9 which indicate that machine learning techniques can effectively be applied to classification of test failures. The results are however not perfect; the random forest classifier is still wrong 10% of the time. Therefore, the classification results should not be considered as an absolute truth, but rather as an indication of what the results might be to guide the debugging.

The second best classifier, decision tree, yielded similar but slightly lower results compared to the random forest. Decision tree and random forest both utilise a tree structure but the difference is the amount of trees used in the algorithms. Algorithms utilising a tree structure have been proven to be effective in classification of root causes and similar results were replicated in this thesis.

Optimising the classifiers by tuning their hyperparameters did not lead to significant performance increases, as can be observed in Table 5.4. This could be because the hyperparameters do not have a great impact on the performance or because we missed the best hyperparameter combinations.

The results presented in Figure 5.2 indicate that the performance of the random forest classifier would not increase notably if more samples were to be added to the dataset. A slight upward trend can be observed as the number of samples approach 10000, but it would require a large increase in the number of samples to yield a significant performance increase.

However, increasing the number of samples greatly improved classification performance when the dataset was small. Our results when using smaller amounts of data are consistent with the results obtained by Chakrabarty et al [4]. They investigated classification of root causes using decision tree and presented an accuracy of 0.84 with 1000 samples, which is very similar to the results yielded by our random forest classifier when using around 1000 samples. This indicates that the amount of data is very important in the beginning, but the importance tapers off after a point. Judging from our results, this point appears to be 6000 samples with 500 from each class.

The worst performing algorithm was the naive Bayes classifier. The model assumes all features are independent, as described in section 3.3.1. Since the result of the naive Bayes was the worst, the features we used have a dependency of each other. The result of our log line abstraction consisted of counting the frequency of messages in log files. The inferior performance of the naive Bayes classifier indicates there exist dependencies between messages in log files. Independent features are rarely observed in reality and there was no exception in our investigation.

The confusion matrix in Figure 5.1 indicated that some root causes were more difficult to classify than others. The confusion matrix indicates that the characteristics of data conversion and RTL bug can be similar. Furthermore, contradicting constraints were misclassified as missing constraints. In this case, the type of root cause was misclassified but the label of lower granularity could still be determined. The classifier also had difficulties classifying OpenCL flaws. OpenCL flaws were represented by 620 samples in the dataset thus the small amount of samples could increase the difficulty of learning the characteristics for this flaw.

When using the random forest classifier to perform classifications with different granularity levels, we observed increases in the accuracy score at every level except the fourth

as can be seen in Table 5.6. The reason for the decrease in accuracy at the fourth level was likely that the sizes of the classes were unbalanced. The difference between level five and level four was that three root causes were grouped together, creating one class that was three times larger than the other classes. At levels three, two and one the accuracy increased, indicating that classification works better with fewer classes.

6.3 Clustering

The results yielded by all clustering algorithms were poor. In Figure 5.5 it is evident that a clustering with AMI and ARI scores around 0.5 is very different from the ground truth. Relying heavily on the output of any of the evaluated clustering algorithms is therefore not advised. Pairing clustering algorithms with visualisation algorithms may however be used to draw additional conclusions about the data based on where test cases are located in the visualisation. For example, in Figure 5.4 the clustering algorithm erroneously identifies a fourth cluster. Without the visualisation the test cases in the fourth cluster would have been treated as having the same root cause, but after inspecting the visualisation it can be determined that they likely do not share a root cause since they are located far from each other.

6.4 Features

Since neither optimising hyperparameters nor increasing the size of the dataset appeared to increase the classification performance, the remaining area of improvement is the transformation from log file to features. We constructed our features using log line abstraction which has been used before and also agreed with our intuition about how to construct the features. When creating the regular expressions used in the abstraction process we tried to limit the amount of system knowledge required. We decided on this more general approach because one reason for using machine learning is that it only requires data to produce good results. Using extensive system knowledge when preparing the data would thus defeat the purpose of using machine learning.

For a completely general approach, the abstraction process should have been performed without any system knowledge. That way, the method and results of this thesis could have been applied to other systems that use log files to track their execution. We tried to automatically create the regular expressions using the LogCluster tool but were not satisfied with the results. In order for that approach to work the contents of the log files would have to be filtered to prevent the tool from identifying unusable patterns. This would however introduce system knowledge into the process, which led us to not investigate a completely general approach further.

Using more system knowledge to construct the features would also be a possibility. Most numerical values in the log files were ignored since it was not apparent what they represented. With more system knowledge it could be possible to extract the numerical values and learn patterns in them. Other parts of the log file that we did not use were the textual representations of different data structures. These were ignored since it was not straightforward how to transform them into features. Successfully transforming these data

structures into features could provide more detailed information about test case executions.

Our approach to feature construction limited the amount of system knowledge required. Thus, our method can easily be applied to other systems that store information about their execution in log files. However, the regular expressions used to construct features have to be modified to match the log messages in the system. It is however difficult to draw any conclusions about what results will be obtained for other systems. The results will depend on how informative the log messages and their frequencies are, which most likely varies between systems. Therefore, our results can only be said with certainty to be applicable to the verification tests performed at Axis. Further investigation would have to be performed to determine the applicability to other types of tests.

6.5 Dimensionality Reduction

For classification, reducing the dimensionality of the feature set using dimensionality reduction algorithms did not improve scores for the classification metrics. For all three clustering algorithms the performance did however improved when the dimensionality was reduced by the PCA algorithm. This indicates that clustering algorithms are more sensitive to the effects of high-dimensional feature spaces compared to classification algorithms. This agrees with the fact that distance, which is the similarity metric used in clustering, is affected by the curse of dimensionality. Longer distances between samples make it harder to determine which are similar enough to belong to the same cluster.

For both clustering and classification, dimensionality reduction methods reduced computation time. The computation time is dependent on the dimensionality of the feature set. Larger feature sets lead to longer computation times. The usage of PCA and the wrapper method `SelectFromModel` is effective when reducing the dimensions of the feature set.

Chapter 7
Conclusion

We have presented a method of automating clustering and classification of test failures. We have shown how textual information in log files can be transformed into a numerical feature vector for input to machine learning algorithms. A comparison of dimensionality reduction methods in conjunction with machine learning algorithms for clustering and classification was performed. The most suitable clustering algorithm was DBSCAN with the dimensionality reduction method PCA, which yielded an AMI score of 0.593 and an ARI score of 0.545. The most suitable classification algorithm was random forest with an accuracy of 0.907 and an F_1-score of 0.913. We also showed how the number of samples in the dataset affects the performance of the classifier.

The DBSCAN and the random forest algorithms were used in the implementation of a tool for automatic clustering and classification of test failures. The tool utilised clustering in conjunction with the visualisation algorithms MDS and t-SNE. Classification in the tool was implemented in conjunction with a confidence level.

From the results of the evaluation process we draw the conclusion that machine learning can be effectively applied to root cause classification. While not perfect, the random forest classifier can be used to quickly get an overview of the root causes present in a failed test run. This overview will hopefully facilitate an efficient distribution of debugging efforts, increasing the speed of the debugging process.

We also conclude that applying machine learning to the clustering problem is not as effective. Our evaluation results suggest that clustering algorithms, while better than a random process, are too inaccurate to be relied upon.

It should be noted that the results of this thesis can only be directly applied to classification of test failures using log files produced by the test benches used by the verification team at Axis. Applying our method to other test benches or other types of tests that produce log files would probably yield similar results, but it is not certain. To determine whether or not this method can be applied a study similar to this one would have to be performed.

7.1 Future work

The evaluated classification algorithms were representatives for all groups of classic classification algorithms. The next step in improving classification performance would be to investigate the application of deep learning. Deep learning methods were not investigated because of the requirement of a large dataset which we did not have access to.

Hyperparameter optimisation for the classification algorithms did not yield significant performance improvements. The hyperparameter ranges used in the searches were limited and not every parameter for the algorithms were included. Further investigation can be performed with all parameters and increased hyperparameter ranges. This may lead to performance improvements but would have a high computational cost.

Our feature construction process relied on system knowledge to extract the important parts of log messages that were combined and used as message representatives whose frequencies could be used as features. The feature construction heavily influenced the performance of the algorithms and potential improvements can be investigated. Investigating a more general approach would be interesting since it would show if machine learning can be applied to the root cause analysis problem in general or if our results were specific to our dataset. Another approach would be to utilise further system knowledge to extract more information from the log files. Textual representations of data structures and numerical values were mostly ignored by us. Extracting them could uncover more patterns in the data and thus improve the performance of both classification and clustering.

The data used in our thesis was limited to log files, but other artefacts, such as waveforms, are also created during test executions. These artefacts could provide additional information about the test failure thus increasing the quality of the feature set. The benefit of using other artefacts could thus be investigated.

Finally, it should be noted that our evaluation was performed using data generated by injecting flaws into designs or test benches to create test failures. Further investigation could be conducted with data produced during real test runs. This would produce more accurate results, but would require the test cases to be properly labelled and stored after each test run.

Bibliography

[1] B. Wile, J. Goss, and W. Roesner. *Comprehensive Functional Verification: The Complete Industry Cycle*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, May 2005.

[2] H. D. Foster. Trends in functional verification: a 2014 industry study. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6. ACM, June 2015.

[3] S. Karlapalem and S. Venugopal. Scalable, Constrained Random Software Driven Verification. In *2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, pages 71–76, Dec 2016.

[4] F. Ye, Z. Zhang, K. Chakrabarty, and X. Gu. Adaptive Board-Level Functional Fault Diagnosis Using Decision Trees. In *2012 IEEE 21st Asian Test Symposium*, pages 202–207, Nov 2012.

[5] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 36–43, May 2004.

[6] H. Lal and G. Pahwa. Root cause analysis of software bugs using machine learning techniques. In *2017 7th International Conference on Cloud Computing, Data Science Engineering - Confluence*, pages 105–111, Jan 2017.

[7] Z. Zhang, X. Gu, Y. Xie, Z. Wang, Z. Wang, and K. Chakrabarty. Diagnostic system based on support-vector machines for board-level functional diagnosis. In *2012 17th IEEE European Test Symposium (ETS)*, pages 1–6, May 2012.

[8] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 465–475, May 2003.

[9] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, Feb 2018.

[10] IEEE Standard for Universal Verification Methodology Language Reference Manual. *IEEE Std 1800.2-2017*, pages 1–472, May 2017.

[11] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[12] P. R. Panda. SystemC: A Modeling Platform Supporting Multiple Design Abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, pages 75–80, New York, NY, USA, 2001. ACM.

[13] G. Rossum. Python Reference Manual. Technical report, Amsterdam, The Netherlands, 1995.

[14] M. C. Cooper, D. A. Cohen, and P. G. Jeavons. Characterising tractable constraints. *Artificial Intelligence*, 65(2):347–361, 1994.

[15] P. Norvig and S. Russel. *Artificial Intelligence: A Modern Approach (3rd Edition)*. Prentice Hall, Upper Saddle River, New Jersey, July 2016.

[16] A. Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Rielly, Gravenstein Highway North, Sebastopol, California, Mar 2017.

[17] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

[18] J. Philip. *The probability distribution of the distance between two random points in a box*, volume 7 of *TRITA / MAT / MA: TRITA*. KTH mathematics, Royal Institute of Technology, 2007.

[19] F. Sebastiani. Machine Learning in Automated Text Categorization. *ACM Comput. Surv.*, 34(1):1–47, Mar 2002.

[20] I. Inza, P. Larrañaga, R. Blanco, and A. J. Cerrolaza. Filter versus wrapper gene selection approaches in DNA microarray domains. *Artificial Intelligence in Medicine*, 31(2):91 – 103, June 2004. Data Mining in Genomics and Proteomics.

[21] P. Somol, B. Baesens, P. Pudil, and J. Vanthienen. Filter- versus wrapper-based feature selection for credit scoring. *International Journal of Intelligent Systems*, 20(10):985–999, Oct 2005.

[22] A. Rocha and S. K. Goldenstein. Multiclass From Binary: Expanding One-Versus-All, One-Versus-One and ECOC-Based Approaches. *IEEE Transactions on Neural Networks and Learning Systems*, 25(2):289–302, Feb 2014.

[23] D. R. Cox. The regression analysis of binary sequences (with discussion). *J Roy Stat Soc B*, 20:215–242, 1958.

[24] S. L. Hamilton and J. R. Hamilton. Predicting in-hospital-death and mortality percentage using logistic regression. In *2012 Computing in Cardiology*, pages 489–492, Sept 2012.

[25] H. Liu, T. Li, L. Chen, S. Zhan, M. Pan, Z. Ma, C. Li, and Z. Zhang. To Set Up a Logistic Regression Prediction Model for Hepatotoxicity of Chinese Herbal Medicines Based on Traditional Chinese Medicine Theory. *Evidence-based Complementary & Alternative Medicine (eCAM)*, pages 1 – 9, 2016.

[26] T. Liu, S. Wang, S. Wu, J. Ma, and Y. Lu. Predication of wireless communication failure in grid metering automation system based on logistic regression model. In *2014 China International Conference on Electricity Distribution (CICED)*, pages 894–897, Sept 2014.

[27] D. G. Kleinbaum and Mitchel Klein. *Introduction to Logistic Regression*, pages 1–39. Springer New York, New York, NY, 2010.

[28] F. S. de Menezes, G. R. Liska, M. A. Cirillo, and M. J. F. Vivanco. Data classification with binary response through the Boosting algorithm and logistic regression. *Expert Systems with Applications*, 69:62 – 73, Mar 2017.

[29] B. E. Boser, I. M Guyon, and V. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM, July 1992.

[30] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, Sep 1995.

[31] N. Deng, Y. Tian, and C. Zhang. *Support vector machines: optimization based theory, algorithms, and extensions*. Chapman & Hall/CRC data mining and knowledge discovery series. Boca Raton : CRC Press, Taylor & Francis Group, [2013], 2013.

[32] J. D. M. Rennie, L. Shih, J. Teevan, and D. R. Karger. Tackling the Poor Assumptions of Naive Bayes Text Classifiers. In *In Proceedings of the Twentieth International Conference on Machine Learning*, pages 616–623, Aug 2003.

[33] P. Domingos and M. Pazzani. On the Optimality of the Simple Bayesian Classifier under Zero-One Loss. *Machine Learning*, 29(2):103–130, Nov 1997.

[34] A. Y. Ng and M. I. Jordan. On Discriminative vs. Generative Classifiers: A comparison of logistic regression and naive Bayes. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14*, pages 841–848. MIT Press, Dec 2002.

[35] W. Loh. Fifty Years of Classification and Regression Trees. *International Statistical Review*, 82(3):329–348, June 2014.

[36] P. E. Utgoff. Incremental Induction of Decision Trees. *Machine Learning*, 4(2):161–186, Nov 1989.

[37] A. L. Garcia-Almanza and E. P. K. Tsang. Simplifying Decision Trees Learned by Genetic Programming. In *2006 IEEE International Conference on Evolutionary Computation*, pages 2142–2148, July 2006.

[38] L. Breiman. Random Forests. *Machine Learning*, 45(1):5–32, Oct 2001.

[39] L. Breiman. Bagging Predictors. *Machine Learning*, 24(2):123–140, Aug 1996.

[40] N. J. Nilson. *Learning Machines: Foundations of Trainable Pattern-classifying Systems*. McGraw-Hill series in system science. McGraw-Hill, 1925.

[41] G. S. Sebestyen. *Decision-making Processes in Pattern Recognition*. ACM monograph series. Macmillan, 1962.

[42] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. A Wiley Interscience Publication. Wiley, 1973.

[43] V. Pestov. Is the k-NN classifier in high dimensions affected by the curse of dimensionality? *Computers & Mathematics with Applications*, 65(10):1427 – 1437, 2013. Grasping Complexity.

[44] J. Wu. *Advances in K-means Clustering: A Data Mining Thinking*. Springer Publishing Company, Incorporated, 2012.

[45] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, March 1982.

[46] C. Levrard. Quantization/clustering: when and why does k-means work? Jan 2018.

[47] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press, Aug 1996.

[48] S. Mai. Density-based algorithms for active and anytime clustering. Sep 2014.

[49] J. H. Ward Jr. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association*, 58(301):236–244, 1963.

[50] L. Maaten and G. Hinton. Visualizing data using t-SNE. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[51] J. B. Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, Mar 1964.

[52] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, Aug 1995.

[53] O. Caelen. A Bayesian interpretation of the confusion matrix. *Annals of Mathematics and Artificial Intelligence*, 81(3):429–450, Dec 2017.

[54] M. Sokolova and G. Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 45(4):427–437, July 2009.

[55] L. Hubert and P. Arabie. Comparing partitions. *Journal of Classification*, 2(1):193–218, Dec 1985.

[56] N. X. Vinh, J. Epps, and J. Bailey. Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. *Journal of Machine Learning Research*, 11(Oct):2837–2854, 2010.

[57] R. Lletí, M. C. Ortiz, L. A. Sarabia, and M. S. Sánchez. Selecting variables for k-means cluster analysis by using a genetic algorithm that optimises the silhouettes. *Analytica Chimica Acta*, 515(1):87 – 100, July 2004. Papers presented at the 5th COLLOQUIUM CHEMIOMETRICUM MEDITERRANEUM.

[58] P. J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53 – 65, Nov 1987.

[59] M. Nagappan and M. A. Vouk. Abstracting log lines to log event types for mining software system logs. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 114–117, May 2010.

[60] L. Mariani and F. Pastore. Automated Identification of Failure Causes in System Logs. In *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*, pages 117–126, Nov 2008.

[61] R. Vaarandi and M. Pihelgas. LogCluster - A data clustering and pattern mining algorithm for event logs. In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 1–7, Nov 2015.

[62] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in Python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[63] J. Bergstra and Y. Bengio. Random Search for Hyper-parameter Optimization. *J. Mach. Learn. Res.*, 13(1):281–305, February 2012.

EXAMENSARBETE Clustering and Classification of Test Failures Using Machine Learning
STUDENT Andy Truong & Daniel Hellström
HANDLEDARE Erik Larsson (LTH), Lars Viklund (Axis Communications AB)
EXAMINATOR Flavius Gruian (LTH)

Kan en dator lära sig att hitta orsaken till varför tester går fel?

POPULÄRVETENSKAPLIG SAMMANFATTNING **Andy Truong & Daniel Hellström**

Att bestämma varför tester går fel görs i nuläget genom att gå igenom miljontals rader av kryptiska loggfiler. Detta arbete undersöker hur maskininlärning kan tillämpas för att automatisera denna process.

Att noggrant undersöka ritningar för att hitta fel i dem är en viktig process inom design av hårdvarukomponenter och kallas för verifiering. De viktiga frågorna i verifiering är att identifiera att något är fel och vad som är orsaken till felet. Målet med verifiering är att identifiera fel i ritningarna så tidigt som möjligt för att slippa onödiga kostnader senare.

I dagsläget görs verifiering genom att komponenters ritningar simuleras och testas för att se hur komponenterna kommer bete sig i verkligheten. Om någonting går fel i ett test så är det viktigt att snabbt komma fram till vad som gick fel och hur det kan åtgärdas. Detta görs genom att manuellt läsa igenom loggfiler som skapas under testexekveringen och leta efter små detaljer som kan ge ledtrådar om vad som gick fel. Att manuellt läsa igenom miljontals rader av information är tidskrävande och kan lätt bli långtråkigt. Det finns därför ett behov av att kunna automatisera denna process.

Lyckligtvis är datorer väldigt bra på att snabbt hantera stora mängder data. Faktum är att en teknik som kallas maskininlärning faktiskt blir bättre ju mer data som finns tillgänglig. Grundkonceptet inom maskininlärning är att låta datorer titta på ett mycket stort antal par av indata och utdata för att lära sig vilka samband som finns. Sambanden kan sedan användas för att koppla ihop ny indata med rätt utdata. I vårt arbete har vi undersökt hur maskininlärning kan användas för att identifiera orsaken till varför tester går fel och för att gruppera liknande tester. Automatisering av denna process leder till att mindre ingenjörstid behöver läggas på arbetet, vilket i sin tur leder till lägre kostnader. Dessutom gör det verifieringsprocessen snabbare vilket leder till att komponenter kan börja produceras tidigare.

Vårt resultat visar att en dator kan lära sig att hitta orsaken till varför tester går fel. Med hjälp av maskininlärning lär sig datorn mönster i loggfiler och kan därefter i nio fall av tio identifiera orsaken till varför ett test går fel. Däremot har en dator problem med att kunna gruppera och hitta likheter mellan testerna.

Vårt arbete visar hur en dator hanterar information i loggfiler för att kunna hitta orsaken till varför tester går fel och kunna gruppera liknande tester. En metod för att kunna omvandla vanlig text till ett språk en dator förstår presenteras. Denna omvandling används sedan i en jämförelse av maskininlärningsalgoritmer. Vårt arbete visar även att resultatet beror på algoritmen som används för identifiering av orsaken till varför tester går fel. Resultatet beror också på hur mycket och vilken information som omvandlas.