in association with

SIEMENS

Erwin Tumbul

# Test case prioritization in limited environment

**Bachelors's Thesis**

to achieve the university degree of

Bachelor of Science

Bachelor's degree programme: Computer Science

submitted to

**Graz University of Technology**

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Roman Kern

Institute of Interactive Systems and Data Science
Head: Univ.-Prof. Dipl.-Inf. Dr. Stefanie Lindstaedt

Graz, August 2020

This document is set in Palatino, compiled with pdfLaTeX2e and Biber.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

# Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present bachelor's thesis.

31.08.2020

Date

Signature

# Abstract

Test case prioritization is a common approach to improve the rate of fault detection. In this scenario, we only have access to very limited data in terms of quantity and quality. The development of an useable method in such a limited environment was the focus of this thesis. For this purpose, we made use of log output and requirement information to create a cluster-based prioritization method. For evaluation, we applied the method to regressions of a device currently in development. The results indicate no impactful improvement, based on the simple and limited metrics used. To show the importance of fault knowledge, we generated a simplified dataset and applied the same prioritization method. With the now existing awareness of faults we were able to evaluate the method using a well established fault-based metric. The results of the generated dataset indicate a great improvement in the rate of fault detection. Despite the restrictions of this limited environment the implemented method is a solid foundation for future exploration.

# Contents

# List of Tables

# List of Figures

# 1 Introduction

In software development, integration testing is an important step between unit testing and system testing. It deals with verifying that multiple separately developed modules work together. In the scope of this thesis, integration testing describes the integration of a device under test (DUT) into a simplified test system.

A common integration test cycle very often involves a regression test, which validates the integrity of non-modified parts of a DUT. Depending on the project size, the test suite and the degree of automation, this may lead to a test time from hours up to days or even weeks. To minimize the duration of regression tests, different solutions have been proposed. Of particular interest in this thesis are test case prioritization (TCP) techniques.

By scheduling the execution of test cases in an order defined by a specific criterion the detection of faults may be maximized in a limited but reasonable time span. However, the effectiveness and the correctness of prioritization techniques depends on the information taken into account. Usually, it is not known whether a test case detects a fault or not before it is executed. Of course, a test engineer can order the execution manually, based on previous experiences. This is a very simple, but not automated form of TCP. Otherwise, only already known information can help establish a ranking. Very often this involves a test coverage report, a requirement-based weight or some similar characteristic.

Machine learning methods, have the potential to lead to promising results, which leads to the purpose of this thesis. It is part of an internal Siemens innovation project to evaluate the application of artificial intelligence in the integration test of safety firmware modules. The development of such safety relevant modules is subject to strict standards, which involve a very throughout testing procedure. To guarantee the safety properties of the

developed modules, every test case has to be executed and documented. This procedure, while mostly automated, is time-consuming. However, for internal evaluation a short test cycle is of advantage.

Most prioritization methods need extensive data about each test case. In the scope of this project we are only aware of the log output of a test case and the validated requirements. Additionally, the existing test framework is currently limited in its ability to extract more data easily.

Therefore, the focus of this thesis lies in the development of a test case prioritization method in an exising and rather limited execution environment.

# 2 Related Work

This chapter introduces the background of this thesis and shows the most important concepts used. Afterwards, the most relevant solutions are presented.

## 2.1 Background

Siemens is an international company founded in 1847 and is by now one of the leading businesses in industrial solutions. Industrial automation and digitalization are the areas of expertise of the Digital Industries (DI) subdivision of Siemens. For this purpose a highly sophisticated automation framework has been created. The development house (DH) Graz plays an important part in the development and innovation in the DI. Many hardware and software solutions are developed and tested at this location.

The development of safety-based solutions is subject to strict standards. Therefore, the testing process is used not only to validate the functionality of a product but all defined safety requirements as well. As Figure 2.1 shows, the integration test is one stage in this process and evaluates a DUT in an integrated environment.

Strict standards require a very meticulous testing phase, which can be very time consuming. Especially sicnce all requirements have to be reviewed, a short test cycle is of advantage during active development. One possible solution to this problem is TCP.

While the test infrastructure is always in development, extensive modification is not always feasible. Therefore, the problem lies in finding an appropriate method which can be used in an already established testing

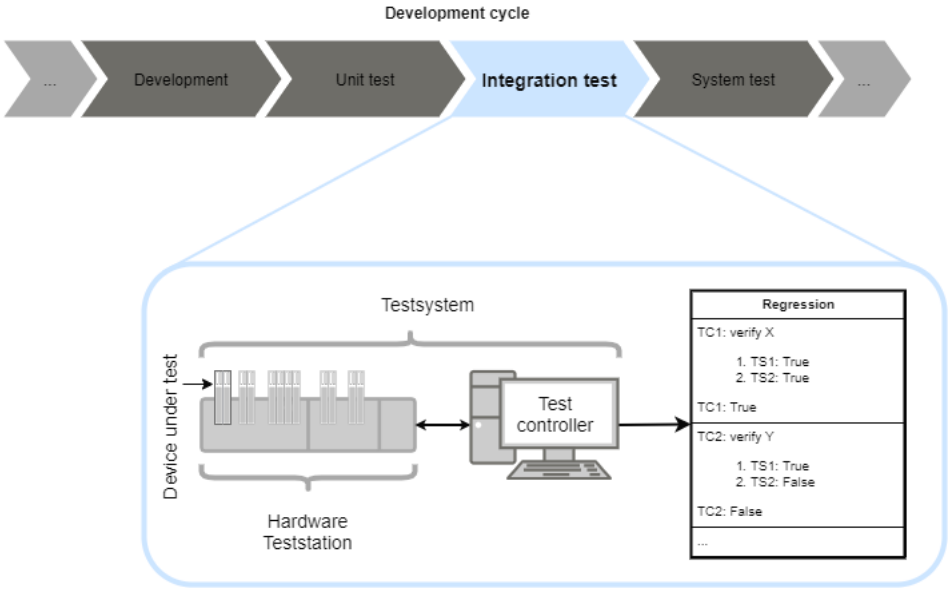


Figure 2.1: A typical development cycle. Every testing step validates requirements and functionality on a different scope. In the integration test some device under test is integrated into a small system, e.g. a test station. A test system is complete together with a test controller, a connected computer in most cases. While specific test cases can be executed solely, usually a regression is preformed. A regression executed a test suit, where each test case verifies one or more requirements and functionality.

environment. There are a few applicable techniques, which are introduced in Section 2.2, but most of them require specific data, like code metrics. At the integration test in the DH Graz there are efforts to measure code metrics, but for now such an approach is not possible.

The focus of this thesis lies in finding an applicable TCP within this constricted environment for practical use in development.

### 2.1.1 Integration Testing

Integration testing describes a common phase in the software development cycle. This type of testing, evaluates the compliance of requirements on

a module-to-module basis. Usually, it is performed after unit testing but before system testing.

A test cycle often involves **regression testing**. It is used to validate the functionality of already developed and tested modules after changes. A regression describes the failure of such a validation. A common problem is the size of regression test suits, which tend to grow rather fast over time. Retesting all test cases may be very time-consuming. Therefore, the execution order of a test suit has to be optimized. This problem can be further classified into three sub-problems [1].

1. **Test suit minimization** aims to remove redundant test cases and minimize the size and therefore, the execution time of the test suit.
2. **Test case selection** seeks to do the same as test suit minimization, with the distinct difference of most such techniques being able to identify modified parts of the module under test. Therefore, test cases which have been modified are more relevant than others and are more likely to be selected.
3. **Test case prioritization** does not search for a subset of relevant test cases to execute, but schedules all test cases depending on different properties. The aim is to find an optimal sequence of test cases, which maximizes the probability of detecting faults.

### 2.1.2 Test Case Prioritization

Rothermel, Untch, Chengyun Chu, *et al.* [2] defined the problem of test case prioritization as follows:

**Definition 1** *Given $T$, $PT$ and $f$, where $T$ is a test suit, $PT$ the set of all possible permutations of and $f$ a mapping from $PT$ to real numbers. Find $T' \in PT | \forall T'' \in PT \wedge T'' \neq T' : f(T') \geq f(T'')$.*

The idea is to find at least one permutation for the test suit $T$, which maximizes a function $f$. The result of applying $f$ to a reordered test suit is also called *award value* and describes the worth of a ordering. While there

are a few possible goals for a prioritization, we focus on increasing the rate of fault detection.

The simplest method of test case prioritization is a reordering based on the knowledge and experience of a test engineer. The idea behind any automated prioritization technique is to emulate this using available data.

### 2.1.3 Text Representation

To make full use of all available data, in particular log output, some form of representation for text data has to be created. A common method for word vectorization is the use of **word embedding**. It describes various techniques in natural language processing, which map the input to an appropriate vector space. The success and effectiveness of these methods, depends very often on the size of the dataset used for training. While already trained models can be found for many common domain languages, this is not the case for this project. Furthermore, building an embedding model requires a very large dataset. Therefore, another approach is necessary.

#### Term Frequency — Inverse Document Frequency (tf-idf)

This method builds upon the idea of weighting the importance of a word to a document in the scope of a collection of documents, also called corpus. As shown in Equation 2.1, the **tf-idf** is defined as the product of the **term frequency** and the **inverse document frequency**.

$$tfidf(t, d, D) = tf(t, d) \cdot idf(t, D) \tag{2.1}$$

The simplest way to define **term frequency** $tf(t, d)$ is to count the raw occurrences of a term $t$ in a document $d$. Other variants include a boolean mapping, the term frequency over document length, logarithmic normalization or augmented normalized term frequency.

$$idf(t, D) = \log \frac{N}{|d \in D : t \in d|} \tag{2.2}$$

The **inverse document frequency** provides a measure of how common a term is over the complete corpus. Equation 2.2 shows how the $idf$ is obtained by taking the logarithm of the document count $N$ over the count of documents $D$ which contain a term $t$. Again, the $idf$ weight can be calculated using different weighting schemes to account for different situations.

### 2.1.4 Clustering

Clustering or cluster analysis describes the grouping of data related to each other. This technique has shown wide use in many areas of data analysis. Figure 2.2 shows a common application of the Centroid-based **k-means** algorithm. This algorithm does well on clearly separated datasets with an even cluster size.

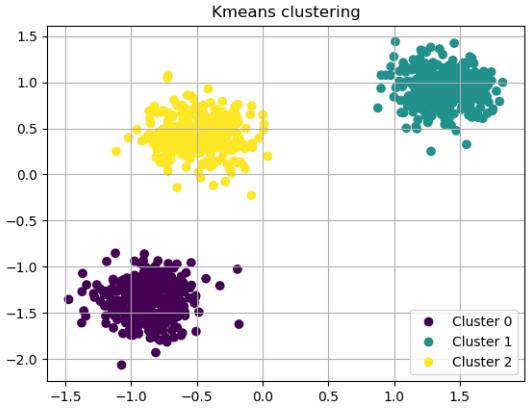


Figure 2.2: Example clustering of randomly generated dataset using the k-means clustering algorithm. The clusters are of similar size and are able to be separated by the algorithm easily.

Within the scope of this project a more irregular dataset can be expected. The following algorithms have shown to be more capable of separating the data.

### Density-based Spatial Clustering of Applications with Noise (DBSCAN)

**DBSCAN** is a very common clustering algorithm, created by Ester, Kriegel, Sander, *et al.* [3]. The algorithm groups together points in a dense neighborhood, depending on the defined radius of neighborhood ($\epsilon$) and minimum number of points in a cluster ($minPts$). Any distance metric can be used for this purpose, but usually the euclidean distance is applied. Points in a low-density region, without nearby neighbors, are classified as outliers. The parameter $m_{pts}$ can usually be derived from a visual analysis or even from the problem statement. $\epsilon$ can be chosen with the help of a sorted k-distance graph, as shown in Figure 2.3. A good value can be shown where the plot shows a change in distance, also described as "elbow" or "knee" [3].

Figure 2.4 shows the difference between **k-means** and **DBSCAN**. Especially arbitrarily shaped clusters can be classified properly, without the need to specify the expected number of clusters. The main disadvantage is the bad handling of clusters with varying density.

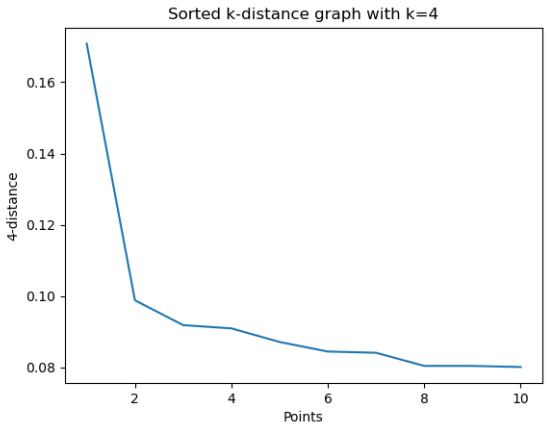


Figure 2.3: Example of a k-distance graph, where k is 4. The elbow shows that the threshold point is around 0.10. This means that any points with a lower value are considered noise and all other points are part of a cluster. This approach does not always guarantee a usable plot, which gives clear indications. Furthermore, detection of this threshold automatically is rather difficult. An interactive approach with the user is recommended.

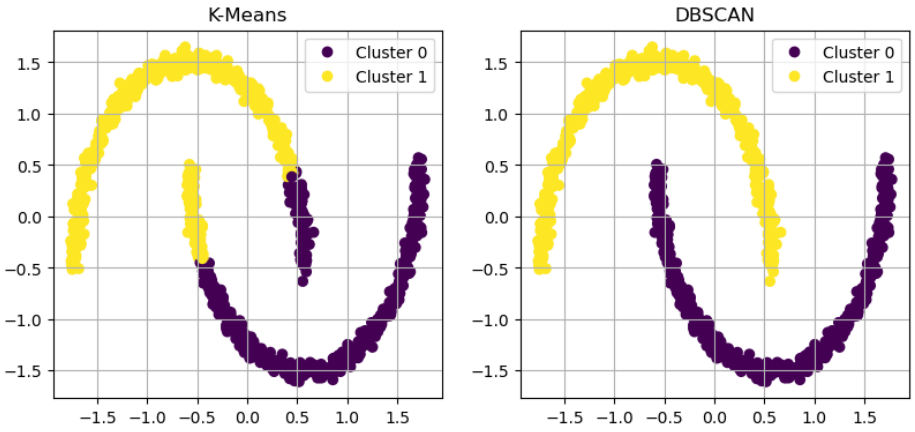


Figure 2.4: Clustering of the randomly generated dataset using **k-means** (left) and **DBSCAN** (right). With this dataset we expect two clusters. **DBSCAN** has been able to handle the unusual shape very well, compared to **k-means**.

### Hierarchical Density-based Clustering (HDBSCAN)

**HDBSCAN** is rather new and improves upon the previously mentioned **DBSCAN** algorithm. It was proposed by Campello, Moulavi, and Sander [4] in 2013 and has been shown to provide a very stable, efficient and correct clustering. The main improvement, compared to **DBSCAN**, is found in the handling of data with varying density clusters. An example of this case, can be observed in Figure 2.5. The algorithm employs a new parameter $m_{clSize}$. It defines the minimum cluster size and allows for the condensation of points which would be split into another cluster otherwise. This simplifies the hierarchy and allows exerting more control over the resulting clustering together with the already established parameter $m_{pts}$. It is even possible to specify a $\epsilon$ threshold to combine **HDBSCAN** and **DBSCAN**.

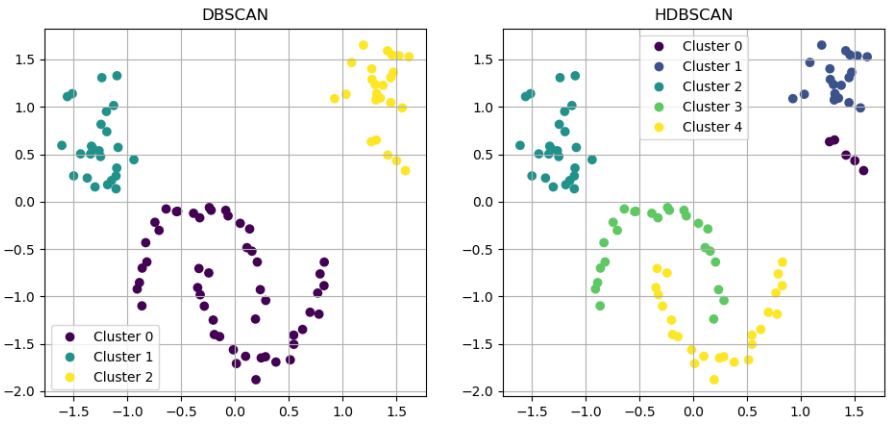


Figure 2.5: Clustering of random generated dataset using **DBSCAN** (left) and **HDBSCAN** (right). While this dataset is rather simple, its shows the advantage of **HDBSCAN** in its result. In this example **DBSCAN** was not able to separate the two interleaving half circles and had problems in finding the small subcluster of "Cluster 2". Better results may be found using a different $\epsilon$ parameter. However, **HDBSCAN** had no such problems and the parameterization was trivial as well.

### 2.1.5 Dimensionality Reduction

High-dimensional datasets are hard to interpret. Due to the "Curse of Dimensionality" clustering is not always effective as well. Therefore, the dimensionality of the data can be reduced, either for visualization purposes or for further analysis. A common technique is Principal component analysis, which is a linear mapping of the data to a low-dimensional space. The aim is to maximize the variance in a few principal components, which represent the original data.

**Uniform Manifold Approximation and Projection (UMAP)** In this project the application of **UMAP** has resulted in a better clustering and therefore, a better prioritization in general. The principle behind this algorithm is the construction of a weighted graph. Using this graph we want to find a low

dimensional representation with a similar topological structure. Parameter-wise we can define *n_components*, *n_neighbours* and *min_dist*, the target dimension, the maximum size of a local neighborhood and the minimal distance between points respectively. We can also use different metrics to further control the resulting reduction.

## 2.2 State of the Art

For the problem of test case prioritization a number of approaches have been developed and evaluated. Rothermel, Untch, Chengyun Chu, *et al.* [2] formally defined the problem of TCP and introduced several methods. Each prioritization method is based on different test execution information. Most future works are based on the finding of this paper.

Rothermel, Untch, Chu, *et al.* [5] evaluate the application of test case prioritization in terms of maximizing the fault detection rate. For this purpose they introduce a metric called "Average Percentage of Faults Detected" (APFD) and evaluate several techniques with it. Their results show that the presented methods can improve the fault detection of test suits. To further evaluate the problem of test case prioritization Elbaum, Malishevsky, and Rothermel [6] evaluated 18 different techniques with three research questions in mind. After extensive experimentation and evaluation their results show that each technique can improve the rate of fault detection.

Walcott, Soffa, Kapfhammer, *et al.* [7] reduced the problem of the prioritization of a test suit within a time constraint to the NP-complete zero/one knapsack problem. This problem can usually be efficiently approximated using a genetic algorithm. Using such an algorithm, has proven to deliver an effective prioritization.

Another possible approach is shown by Tonella, Avesani, and Susi [8]. They have evaluated the incorporation of user knowledge through a machine learning algorithm, called case-based ranking (CBR). Metrics like code coverage can be used in CBR to approximate a ranking and specific user knowledge can be applied if either not enough data is available or if there are any contradictions.

Yoo, Harman, Tonella, *et al.* [9] proposed the usage of clustering in their method. The purpose is to ease the actual prioritization process, by grouping similar test cases in terms of runtime behavior together. Therefore, minimizing the workload of a test engineer. Additionally, they have shown that clustering alone, in an automated process, might yield a considerable improvement.

Carlson, Do, and Denton [10] presented a similar method. They have demonstrated, through the usage of clustering, based on the similarity of common properties, an effective prioritization. In their study, they have made use of information like code coverage, code complexity, historic fault information and a combination of these.

Opposed to the previously mentioned techniques, Arafeen and Do [11] incorporated requirement-based information into the testing process. Their method includes the clustering of requirements, the prioritization of test cases for each requirement cluster and a final reordering, based on requirement priority. The results of their evaluation demonstrate the impact of requirements-based clustering in test prioritization.

# 3 Method

This project is based on the idea of applying test case prioritization in a very limited environment. One of these limitations is insufficient data. Most machine learning methods require a large amount of data during training to produce stable and reliable results. Furthermore, we are only aware of the test case output and the names of covered requirements. The lack of code metrics impedes the usage of many established solutions shown in Section 2.2. Besides, the modification of an established test system is not always feasible, due to its size and complexity. Therefore, the focus lies on the development of a prioritization method applied in the scope of a very limited environment.

## 3.1 Concept

The input data for a prioritization consists of a past regression, where each test case has an output log and a list of requirements. When a module is tested, each test case is executed separately on the module. The module resets its state after the execution of any test case with either result.

Three assumptions follow:

**Assumption 1** *Test cases are independent of each other.*
**Assumption 2** *Test cases may validate the same requirements.*
**Assumption 3** *The result of a test case is reflected in its output.*

Figure 3.1 shows the general workflow of the method. The preprocessing step can be executed once during the first iterations or even beforehand. It involves

- date-time handling,

- text aggregation,
- text transformations like
  - conversation to lowercase,
  - number handling,
  - white space handling and
  - stop word and special character removal.


Figure 3.1: This is the general workflow of the presented prioritization method. The input consists of a list of test cases and the output is composed of the prioritized order of test cases. The preprocessing step involves common text processing methods to prepare for the usage in automated processes. The prioritization step in this graph is where we perform the prioritization using different input models.

Based on above assumptions, we can define the problem statement for a cluster-based approach:

**Definition 2** *Given a set of test cases $T$, we want to find a partition of $n \in \mathbb{N}$ test case subsets $T_1$ to $T_n$ such that every test case $t_1$ to $t_i$ with $i \in \mathbb{N}$ is exactly once in one of these $n$ clusters. Each cluster $T_n$ contains test cases more simiar to each other.*

Given a clustered test suite, we can assign each cluster a priority based on different cluster-wise metrics. Depending on this priority, we can reorder the execution of the test suite to maximize or minimize the given metric.

The use of mainly text as input, requires the application of text vectorization. We use **tf-idf** to create a text representation.

Depending on the number of features per test case output, very large and often sparse matrices are the result. To visualize high-dimensional data, the application of dimensionality reduction techniques is necessary. Furthermore, the use of distance metrics for clustering of high-dimensional

data may lead to problems because of the "Curse of dimensionality". The application of **UMAP** has shown the best results out of the tested methods.

In our tests we tried **k-means**, **DBSCAN** and **HDBSCAN** for clustering, where the last has shown to acquire the best results in terms of viability and consistency.

### 3.1.1 Process

In this section we present the main workflow of cluster-based prioritization. Afterwards follow the three input models and their specifics.

Figure 3.2 shows the prioritization workflow, which can be split into following steps:

1. Vectorize input data.
2. Reduce dimensionality.
3. Apply clustering.
4. Prioritize test cases within clusters.
5. Prioritize clusters.
6. Select test cases.

Each model uses a different type of input data. This input is vectorized in step 1 using **tf-idf**. Step 2 describes the reduction of the high-dimensional **tf-idf** matrix using **UMAP**. These steps are in general the same, except for the difference in data used.

In step 3 we apply **HDBSCAN** to cluster the input data. To improve the resulting clustering we employ an interactive approach for fine-tuning the parametrization. The result is plotted and the user may accept or decline the result. If declined, the parameters can be specified again until an acceptable result is reached.

In step 4 and 5 we optimize the order of test cases and the order of clusters. This reordering assigns a priority to test cases and clusters and ranks them descending. Which step is performed first has no actual impact, but in the scope of this implementation the above specified order applies.

In the last step we select the test cases from the prioritized clusters with following Assumption.

**Assumption 4** *The top ranked test cases within high-priority clusters are more likely to find faults.*

We compare two methods. In the first, which we call *SingleSample*, we simply take the best test case from each cluster in a round-robin fashion. In the second, called *PercentageSample*, we pick the best test cases of each cluster by percentage. We take the most from the first cluster and reduce the amount for each following cluster. This step is again repeated in a round-robin movement. The idea behind both is based on the test case selection in the method created by Arafeen and Do [11].

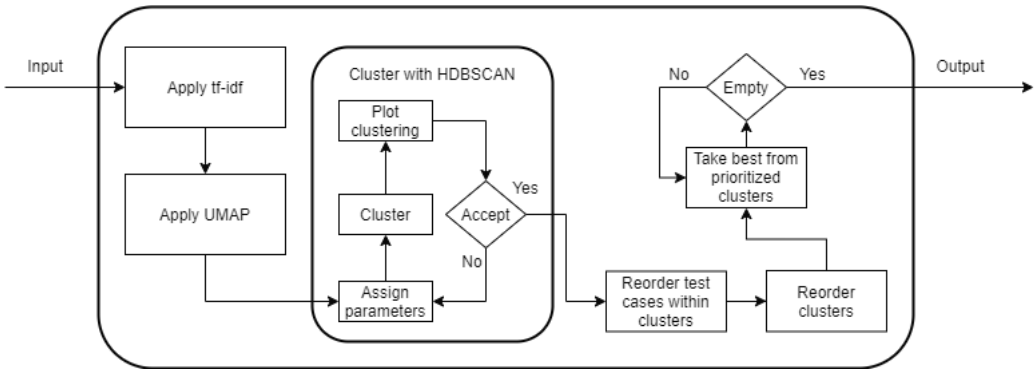


Figure 3.2: The workflow of a cluster-based prioritization. First the input is vectorized using **tf-idf** and then the dimensionality is reduced using **UMAP**. Afterwards the data is clustered using **HDBSCAN**. The clustering requires an interactive approach to define the parameters of the **HDBSCAN** algorithm. To make this easier for the user, the resulting clustering is shows afterwards in a plot. If acceptable the user confirms and continues with the chosen prioritization model. Afterwards the clusters and the test cases within are prioritized. The last step is the selection of test cases from each cluster until there are none left.

### 3.1.2 Input Models

In the scope of this project we differ between two types of inputs, which define following models.

### Data Model: $M_D$

**Assumption 5** *Test cases with similar output, are more likely to find the same errors.*

Due to Assumption 3.1, the use of output data is constricted in its application. Since the result is reflected in the output, a prioritization might be biased to a specific result, as can be seen in Figure 3.3. To handle this potential problem, the prioritization has to be created with a "normalized" regression. Therefore, we will create a prioritization based on a completely successful regression.

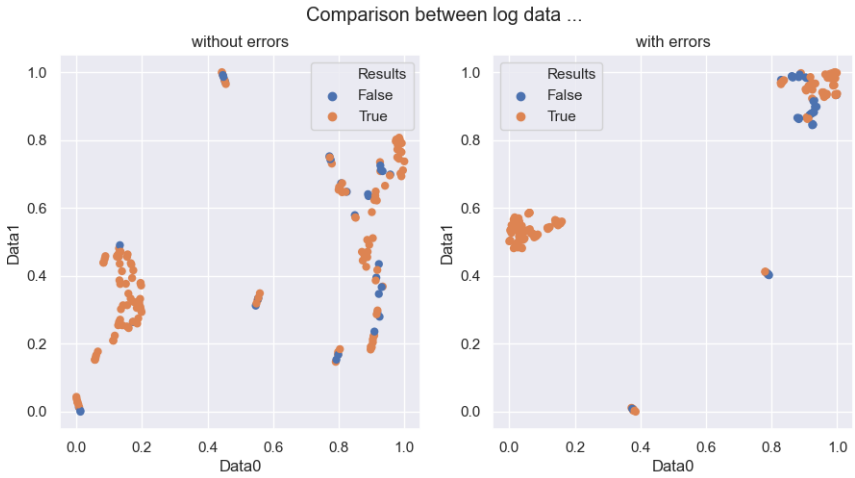


Figure 3.3: Comparison of vectorized log output of a regression without errors (left) and with errors (right). The right plot shows how the failed test cases are grouped together better than on the left. While this may be the desired result in some cases, here it is not. We want to cluster the test case outputs based on the similarity of their structure not their result, because it might lead to bias.

### Requirement Model: $M_R$

**Assumption 6** *Test cases with similar requirements, are more likely to find the same errors.*

We make use of the requirement data by assigning each test cases its list of requirements in text form. Then by clustering this data, we can find test cases with similar requirements. This assumption is based on Assumption 3.1. Since this prioritization is independent of the output, we even are able to apply it beforehand.

## 3.2 Implementation

As most data science application we use Python[1] together with following libraries:

- Numpy[2]
- Pandas[3]
- Sklearn[4] [12]
- Umap[5] [13]
- Hdbscan[6] [14]
- Matplotlib[7]
- Seaborn[8]

### 3.2.1 Architecture

The extension of an established test framework is not easy. Including completely new subsystems and fitting them into the execution pipeline even more so. Therefore, we will develop an external framework which can be automated easily using existing solutions. In this approach, we can take advantage of Python's flexibility and ease of use, to create an effective test case prioritization.

[1] Python 3.8.3
[2] numpy 1.18.4
[3] pandas 1.1.0
[4] scikit-learn 0.23.1
[5] umap-learn 0.4.6
[6] hdbscan 0.8.26
[7] matplotlib 3.2.1
[8] seaborn 0.10.1

| Name | Description | Remarks |
|---|---|---|
| Testcase | Name of a test case. | |
| Result | Result of a test case. | Either 0 (failure) or 1 (success); used for evaluation. |
| Duration | Duration of a test case. | In minutes. |
| Coverage | Percentage of requirements covered. | Derived from requirements list. |
| Data | Output log from test case execution. | Needs text preprocessing. |
| Requirements | List of covered requirements of a test case. | Needs text preprocessing. |

Table 3.1: A dataset consists of above defined attributes in form of columns. The *Result* and *Duration* are extracted from *Data*. In this project everything except for the requirement data is queried from a database. The requirements are extracted using a different method and are added to each regression of this test suit afterwards. In the same step the requirement coverage is calculated.

The input data consists of a past regression saved in form of a CSV file, pulled from a database.

Table 3.1 shows the data required in an input file. The pre-processing step can be performed here during the creation of the dataset using a Python script. This separate script queries the data from the database and aggregates it after applying the methods shown in Section 3.1.

We create a class called **ClusterTCP** which is based on the Sklearn uniform API. To conform to this API we inherit from *BaseEstimator* and *TransformerMixin*. Appendix A.1 contains the bare-bones class structure, without implementation.

We differ between the fitting, which involves the vectorization, dimensionality reduction and clustering step, and the transformation. In the transformation we apply a user defined function, which reorders the fitted data. While an interactive fitting is recommended to find the best clustering for any input, the parameters can be requested and saved by the user for direct initialization afterwards. These parameters can be defined in the constructor, or with a method. Since we inherited from *BaseEstimator*, methods for

getting and setting parameters are already implemented. If the *interactive* flag is set, any clustering parameters are ignored and the user is required to enter them in the console when requested. The resulting clustering is plotted then, as shown in Figure 3.5, and the user may accept or decline in the console again. If declined, the cluster step is repeated until acceptance.

For evaluation purposes we define a score function as well which returns metrics further discussed in Chapter 4 and implement a plotting method for visual inspection. For visual evaluation of the prioritization **ClusterTCP** offers a *show* method, were we display the reordered test cases in an event plot with their results as shown in Figure 3.4.

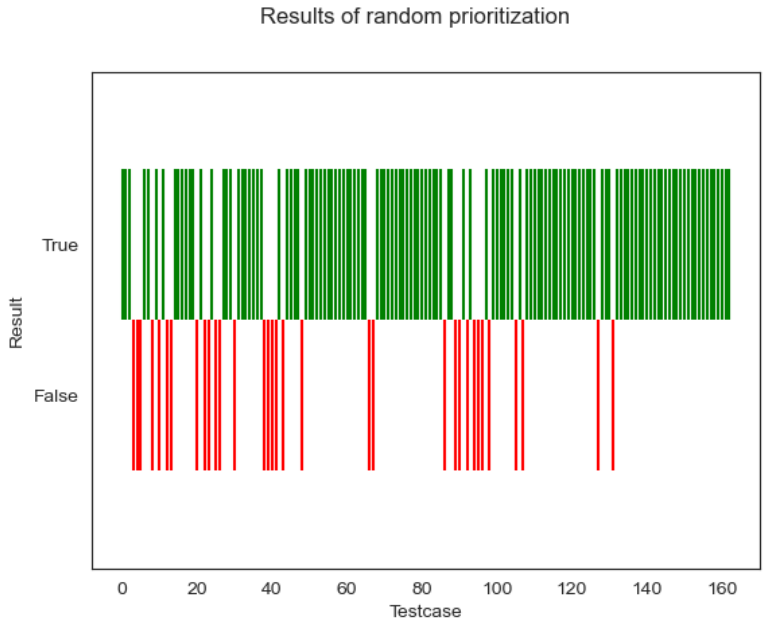


Figure 3.4: Example of the event plot shown by the *show* method. This kind of plot allows the user to immediately realize the effectiveness of the constructed test case prioritization. Additionally, comparison between different results is very easy as well.

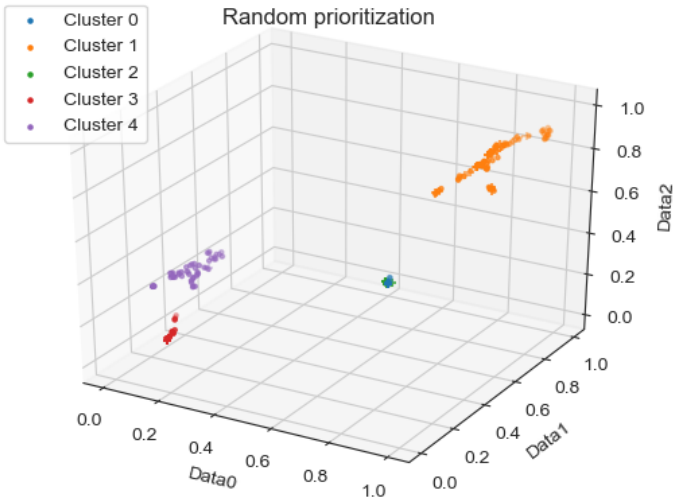


Figure 3.5: Example of the created clustering in the *fit* method using the data model $M_D$. This plot allows the user to fine-tune the parameters for effective test case prioritization. The plot itself is interactive as well and can be rotated, which allows a better analysis.

# 4 Evaluation

The focus of this thesis lies in the application of TCP in a limited environment. In this statement included is the lack of meaningful data usually used in other prioritization techniques presented in Section 2.2. But the amount of data is not the largest problem here, because the data itself is rather limited in its application.

In this evaluation we make use of the four datasets shown in Figure 4.1, which have been generated during the test phase of a module currently in development. All four datasets have been generated using the same test suit, therefore every regression was executed in the same order. The only difference is the hardware version used. Each version induced different faults, which led to the failure of different test cases.

| Datasets | Test cases | Errors | Duration |
|---|---|---|---|
| successful | 163 | 0 | 39010 |
| failed_01 | 163 | 33 | 40141 |
| failed_02 | 163 | 28 | 44847 |
| failed_03 | 163 | 21 | 45861 |

Table 4.1: This table shows the datasets used for the method evaluation, where each dataset is a regression of the same test suit. Therefore, the number of test cases is the same in each case. Since in each regression another version has been used the number of errors is different and with that the regression duration (in minutes) as well.

Furthermore, a traceability matrix, mapping each test case to its covered requirements, is available as well and can be found in Appendix A.2.

For comparison, we will evaluate the completely same TCP with a simplified and generated datasets, shown in Table 4.2. This dataset is modeled after the

real data and its structure. The main difference are the randomly assigned faults, to show the impact this knowledge can have during evaluation. Appendix A.5 shows a sample of the dataset for reference.

| Datasets | Test cases | Errors | Faults | Duration |
|---|---|---|---|---|
| successful | 1000 | 0 | 0 | 59669 |
| failed_generated | 1000 | 207 | 5 | 69662 |

Table 4.2: These datasets have been generated by choosing randomly from a pool of values. While the data is kept very simple, the structure is based on the real datasets above. Furthermore, we assigned to each failed test case a fault. To simulate a common order of execution, where test cases with similar validation are executed nearby each other, we assigned the faults in uniform blocks over the whole regression.

## 4.1 Metrics

A very common measure for the evaluation of TCP methods is the Average Percentage of Faults Detected (APFD) introduced by Rothermel, Untch, Chu, *et al.* [5]. This measure gives a score between 0 and 100, where a higher number shows a higher fault detection rate. But it is based on the relationship between faults and test cases. In our environment we are not aware of any faults, only the test case results. To show the advantages of fault knowledge in this evaluation we will apply the APFD to the generated dataset.

To evaluate the applied TCP on the real datasets, we use following simplified metrics and compare with the scores of the original execution order shown in Table 4.3 and Table 4.4. For the generated dataset we do the same with the addition of using the APFD as well.

- Accuracy: A score to show the correctness of the TCP, compared to a "perfect" prioritization, in terms of executing any failed test cases first. Equation 4.1 defines the score, where $T$ is a list of prioritized test cases, $\hat{T}$ is a list of perfectly prioritized test cases and $N$ is the number

of test cases, which has to be the same for both inputs.

$$accuracy(T, \hat{T}) = \frac{1}{N} \sum_{i=0}^{N-1} 1(\hat{T}_i = T_i) \tag{4.1}$$

- Coverage: The fraction of test cases which cover all existing errors of a execution order. Equation 4.2 shows the calculation of this score, where $T$ is a list of prioritized test cases, $i$ is the index of the last failed test case and $N$ is the number of test cases.

$$coverage(T) = \frac{T_i}{N} \tag{4.2}$$

- APFD: The average percentage of faults detected, requires knowledge about the existing faults. It is defined in Equation 4.3, where $n$ is the number of test cases, $m$ the number of faults and $TF_{m_i}$ is the index of the test case which found a fault $m$ first.

$$APFD = 1 - \frac{TF1_i + TF2_i + \ldots + TF_{m_i}}{n \cdot m} + \frac{1}{2n} \tag{4.3}$$

| Datasets | Coverage | Accuracy |
|---|---|---|
| failed_01 | 0.88 | 0.68 |
| failed_02 | 0.96 | 0.68 |
| failed_03 | 0.83 | 0.80 |

Table 4.3: Scores of the originally executed, real regressions.

| Datasets | Coverage | Accuracy | APFD |
|---|---|---|---|
| failed_generated | 0.99 | 0.67 | 0.60 |

Table 4.4: Scores of the originally executed, generated regressions.

## 4.2 Results

This section shows the results of the developed test case prioritization method applied to the above defined datasets. Table 4.5 and Table 4.7 show

the results of the data model $M_D$ and Table 4.6 and Table 4.8 show the results of the requirement model $M_R$.

For both models we applied different reorder combinations, where the left letter describes the data used to prioritize the clusters and the right letter the test cases within each cluster. *D* stands for *Duration* and *R* for *Requirement coverage*. Furthermore, we applied the two selection methods *SingleSample* and *PercentageSample* defined in Section 3.1.1. For each result we observe the two metrics *coverage* and *accuracy*. Additionally, the results of the generated dataset are evaluated using the *apfd*.

Appendix A.4 contains the plotted results of each dataset for both models.

| | SingleSample | | PercentageSample | |
|---|---|---|---|---|
| Dataset | coverage | accuracy | coverage | accuracy |
| *failed_01* | | | | |
| D/D | 0.99 | 0.69 | 0.99 | 0.68 |
| D/R | 0.99 | 0.69 | 0.99 | 0.68 |
| R/D | 0.99 | 0.69 | 0.81 | 0.73 |
| R/R | 0.99 | 0.69 | 0.81 | 0.73 |
| *failed_02* | | | | |
| D/D | 0.96 | 0.73 | 0.96 | 0.69 |
| D/R | 0.96 | 0.73 | 0.96 | 0.69 |
| R/D | 0.96 | 0.74 | 0.97 | 0.72 |
| R/R | 0.96 | 0.74 | 0.97 | 0.72 |
| *failed_03* | | | | |
| D/D | 0.94 | 0.82 | 0.66 | 0.75 |
| D/R | 0.94 | 0.82 | 0.66 | 0.75 |
| R/D | 0.94 | 0.82 | 0.71 | 0.75 |
| R/R | 0.94 | 0.82 | 0.71 | 0.75 |

Table 4.5: Results of $M_D$. We used three regressions and applied four reorder combinations. *D* stands for *Duration* and *R* for *Requirement coverage*. In terms of cluster parameterization, we applied $m_{pts} = 7$ and $m_{clSize} = 10$.

| | SingleSample | | PercentageSample | |
|---|---|---|---|---|
| Dataset | coverage | accuracy | coverage | accuracy |
| *failed_01* | | | | |
| D/D | 0.90 | 0.75 | 0.98 | 0.79 |
| D/R | 0.90 | 0.75 | 0.98 | 0.79 |
| R/D | 0.90 | 0.75 | 0.99 | 0.79 |
| R/R | 0.90 | 0.75 | 0.99 | 0.79 |
| *failed_02* | | | | |
| D/D | 0.96 | 0.71 | 0.98 | 0.71 |
| D/R | 0.96 | 0.71 | 0.98 | 0.71 |
| R/D | 0.96 | 0.71 | 0.99 | 0.72 |
| R/R | 0.96 | 0.71 | 0.99 | 0.72 |
| *failed_03* | | | | |
| D/D | 0.90 | 0.78 | 0.97 | 0.79 |
| D/R | 0.90 | 0.78 | 0.97 | 0.79 |
| R/D | 0.90 | 0.78 | 0.93 | 0.80 |
| R/R | 0.90 | 0.78 | 0.93 | 0.80 |

Table 4.6: Results of $M_R$. We used three regressions and applied four reorder combinations. *D* stands for *Duration* and *R* for *Requirement coverage*. Here we applied a cluster parameterization of $m_{pts} = 2$ and $m_{clSize} = 2$. The requirements data can be clustered rather well and is clearly separable.

| | SingleSample | | | PercentageSample | | |
|---|---|---|---|---|---|---|
| Dataset | coverage | accuracy | apfd | coverage | accuracy | apfd |
| *failed_generated* | | | | | | |
| D/D | 1.00 | 0.67 | 0.77 | 1.00 | 0.67 | 0.89 |
| D/R | 1.00 | 0.67 | 0.77 | 1.00 | 0.67 | 0.89 |
| R/D | 1.00 | 0.67 | 0.78 | 0.99 | 0.68 | 0.92 |
| R/R | 1.00 | 0.67 | 0.78 | 0.99 | 0.68 | 0.92 |

Table 4.7: Results of $M_D$ using the generated dataset. *D* stands for *Duration* and *R* for *Requirement coverage*. The parameterization was performed with $m_{pts} = 2$ and $m_{clSize} = 2$.

| | SingleSample | | | PercentageSample | | |
|---|---|---|---|---|---|---|
| Dataset | coverage | accuracy | apfd | coverage | accuracy | apfd |
| *failed_generated* | | | | | | |
| D/D | 0.42 | 0.79 | 0.83 | 0.98 | 0.75 | 0.77 |
| D/R | 0.42 | 0.79 | 0.83 | 0.98 | 0.75 | 0.77 |
| R/D | 0.42 | 0.79 | 0.83 | 1.00 | 0.59 | 0.42 |
| R/R | 0.42 | 0.79 | 0.83 | 1.00 | 0.59 | 0.42 |

Table 4.8: Results of $M_R$ using the generated dataset. *D* stands for *Duration* and *R* for *Requirement coverage*. The parameterization was performed with $m_{pts} = 2$ and $m_{clSize} = 2$.

| Datasets | Coverage | Accuracy |
|---|---|---|
| $failed_0 1$ | | |
| Original | 0.88 | 0.68 |
| Best $M_D$ | 0.81 | 0.73 |
| Best $M_R$ | 0.90 | 0.75 |
| $failed_0 2$ | | |
| Original | 0.96 | 0.68 |
| Best $M_D$ | 0.96 | 0.74 |
| Best $M_R$ | 0.96 | 0.71 |
| $failed_0 3$ | | |
| Original | 0.83 | 0.80 |
| Best $M_D$ | 0.66 | 0.75 |
| Best $M_R$ | 0.90 | 0.78 |

Table 4.9: Here we compare the best results of the real datasets with the original execution order. The results are approximately around the original scores.

| Datasets | Coverage | Accuracy | APFD |
|---|---|---|---|
| $failed_generated$ | | | |
| Original | 0.99 | 0.67 | 0.60 |
| Best $M_D$ | 0.99 | 0.68 | 0.92 |
| Best $M_R$ | 0.42 | 0.79 | 0.83 |

Table 4.10: This table shows the scores of the best results compared to the original. For both models the APFD shows a great improvement. $M_R$ even improved the *Coverage* and *Accuracy* by a large margin.

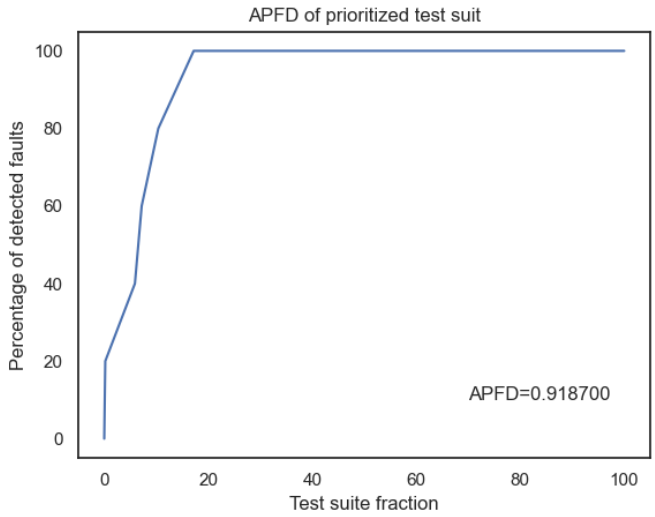


Figure 4.1: The curve of this figure represents the cumulative percentage of faults detected over the life of a test suit. The area under this curve depicts the APFD. This graph shows how we have found all faults after the execution of roughly 20% of the test suit.

## 4.3 Discussion

Overall, the results do not show a great improvement using the real dataset. Furthermore, Table 4.5 and Table 4.6 show that the choice of reorder combination does not seem to make a large difference in most cases.

$M_D$ performed better on the dataset with the least errors using the *PercentageSample* method in the last step. Compared to that, $M_R$ performed better on the same dataset using the *SingleSample* method.

While we could conclude that above observations follow a pattern, there is simply not enough data in this stage of development. What we do know is that a TCP is certainly possible, even in this kind of limited environment, but with limited results. There is still room for improvement in terms of data acquisition. With more data better results may follow, and with more diverse data more methods may be applied.

But as shown in Appendix A.4, if the TCP is time-limited better results may be achieved than using the original execution order. Furthermore, since we are not aware of any faults, we may already have found all of them due to the TCP without being aware of it. This disadvantage has a large impact in the evaluation of this method.

Especially, when comparing the results in Appendix A.4 with the original execution order in Appendix A.3, we can see how the errors are distributed. A test suit usually consists of groups of related test cases. With such an execution order, we will always find faults in these groups in the same sequence. Using a dedicated TCP we should be able to find multiple faults earlier.

The usage of the generated dataset with knowledge of faults, confirms this assumption. In Table 4.7 and Table 4.8 we see similar results in terms of *Coverage* and *Accuracy*, but the APFD shows a great improvement in general. Table 4.10 shows how $M_R$ even improves upon the *Coverage* and *Accuracy* by a great margin.

# 5 Conclusion

There exist quite a few test case prioritization techniques. Most of them depend on different code metrics, which are not available in the scope of this project. To deal with such a limited environment, we applied a few common machine learning methods.

First, we made use of the log output of each test case, assuming that test cases with similar output find similar faults. The application of this method is quite limited, since we need to use an unbiased regression as input. The second approach is based on the relation between test case and covered requirements.

While the results do not show a great improvement based on the simplified metrics used, we can observe an improved distribution of failed test cases. Especially using the requirement model, we can assume an increased likelihood of finding faults. We verified this assumption using a simplified test suit with incorporated fault knowledge we generated. By using the well established APFD metric, we were able to show a large improvement in terms of fault detection after the application of the developed test case prioritization.

To allow for easy usage and further experimentation, we made full use of Python and its flexibility, by implementing a Sklearn-based API. This approach not only makes future modification, but the incorporation into the existing test system rather easy.

In conclusion, a good foundation for further exploration has been created by providing the basic know-how and a simple API. Building upon this groundwork, it should be possible to improve the developed prioritization in future works.

## 5.1 Future Work

The first and essential step is improving the data used for the prioritization. By making use of code metrics, we would be able to make use of already established test case prioritization methods. Furthermore, using the generated dataset we were able to show the importance of fault knowledge. The incorporation of fault data, allows for easy and comparable evaluation of further developed methods.

Another problem is the parametrization of any applied machine learning method. While an interactive workflow is a valid approach, automation is always the next step in any test system. With an automated system the optimization of specific metrics is possible as well.

# Appendix

## A.1 ClusterTCP Class

```
class ClusterTCP(BaseEstimator, TransformerMixin):
    def __init__(self, target_feature='Requirements',
       reorder_f=None, target_components=3, minPts=None
       , minClusterSize=None, interactive=False):
        # Initialize parameters

    def fit(self, X, y=None):
        # Input and parameter validation

        # Apply vectorization

        # Apply dimensionality reduction

        # If interactive flag is set
            # Perform clustering and return
        # else
            # Perform interactive parameter estimation
               and clustering
            # If accepted
                # return
            # else repeat

    def transform(self, X, y=None):
        # Check if fitted

        # Input and parameter validation

        # Apply reorder function defined by user

        # Return transformed data

    def fit_transform(self, X, y):
        # First call fit and then transform X and
```

```
        return

    def plot_clusters(self, X, title='Cluster plot', ax
        =None):
        # Plot X if fitted

    def show(self, title):
        # Print and display resulting prioritization

    def score(self):
        # Return score
```

## A.2 Traceability Matrix

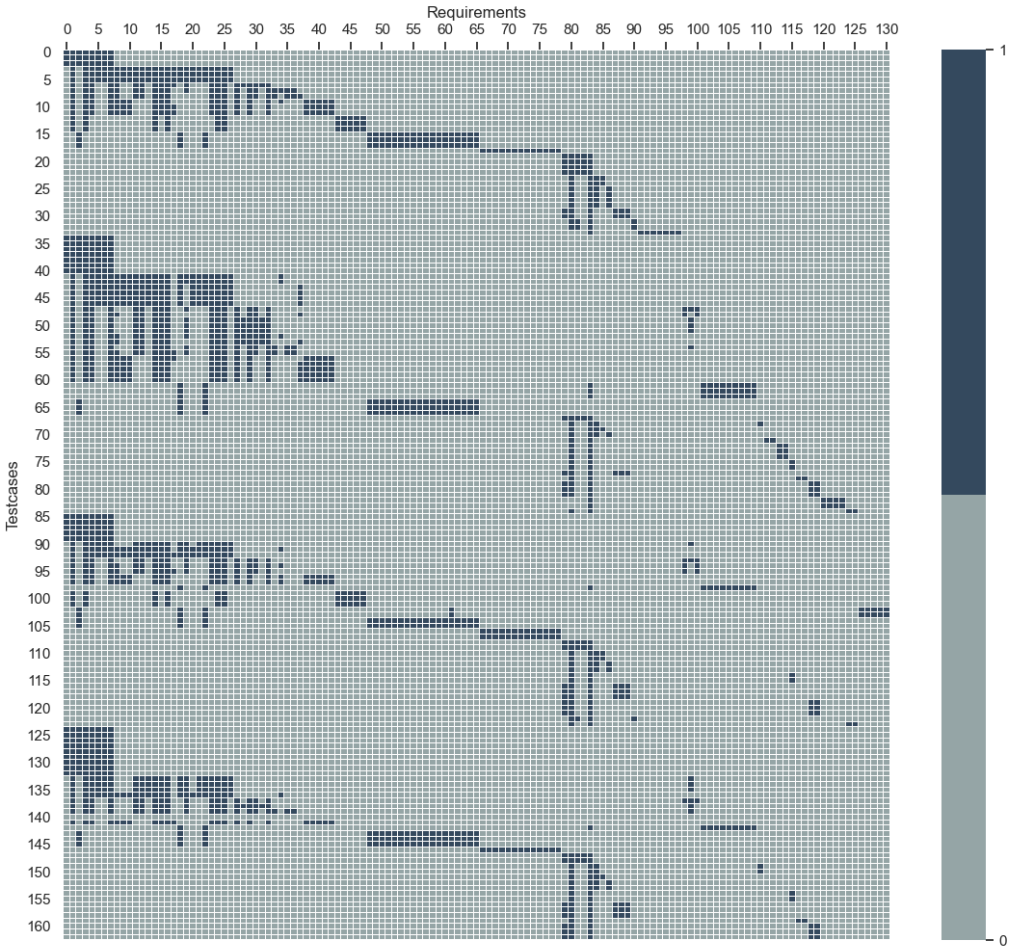


Figure A.1: This traceability matrix describes the relationship between test cases and requirements, where each test case covers a number of requirements. A test case covers a requirement if the cell dark blue (1). The opposite is the case if a cell is light blue (0). This matrix applies to all datasets used in Chapter 4. The names of test cases and requirements have anonymized.

## A.3 Original

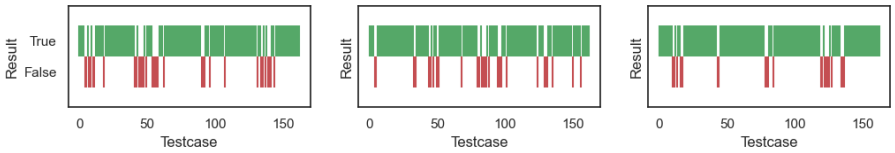


Figure A.2: The original execution order with results for *failed_01* (left), *failed_02* (middle), *failed_3* (right).

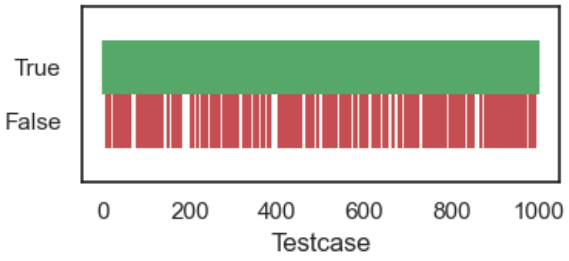


Figure A.3: The original execution order with results for *failed_generated*.

## A.4 Results

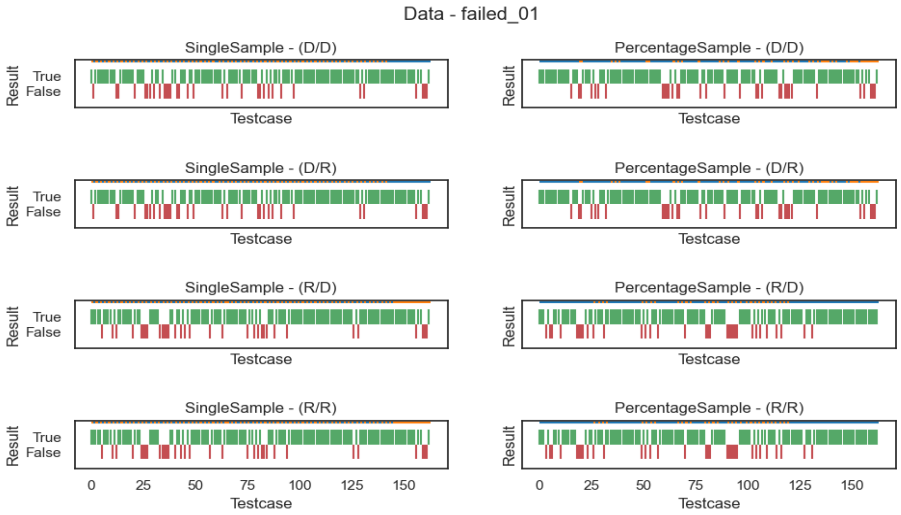


Figure A.4: Results of TCP applied to Model $M_D$ using dataset *failed_01*.

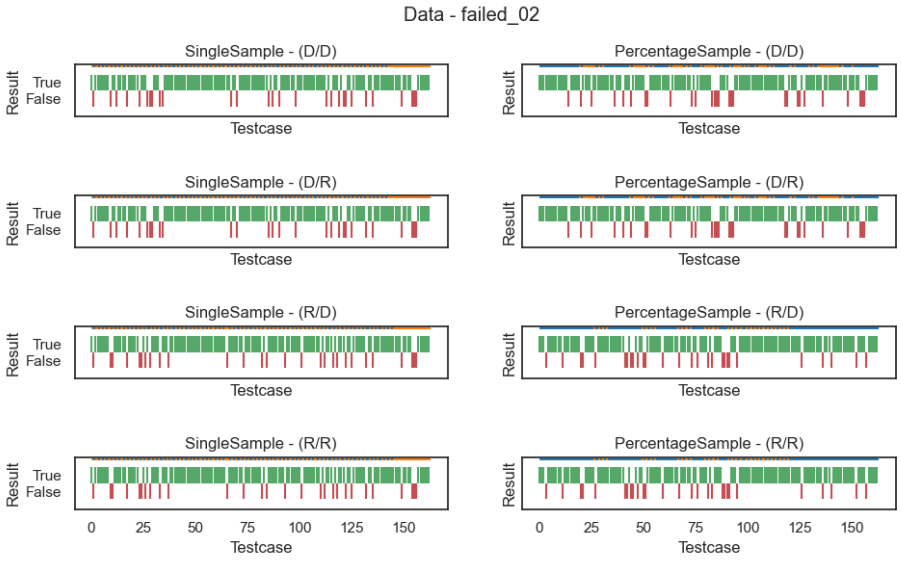


Figure A.5: Results of TCP applied to Model $M_D$ using dataset *failed_02*.

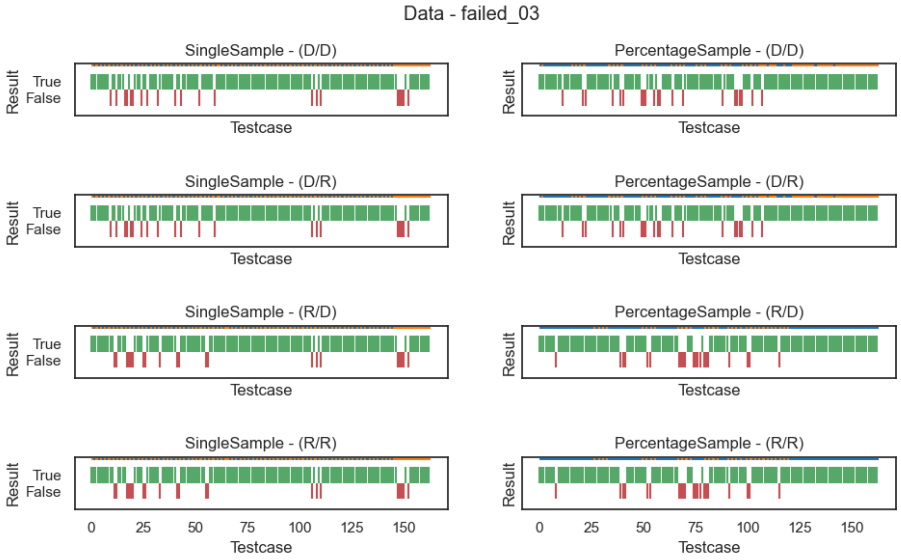


Figure A.6: Results of TCP applied to Model $M_D$ using dataset *failed_03*.

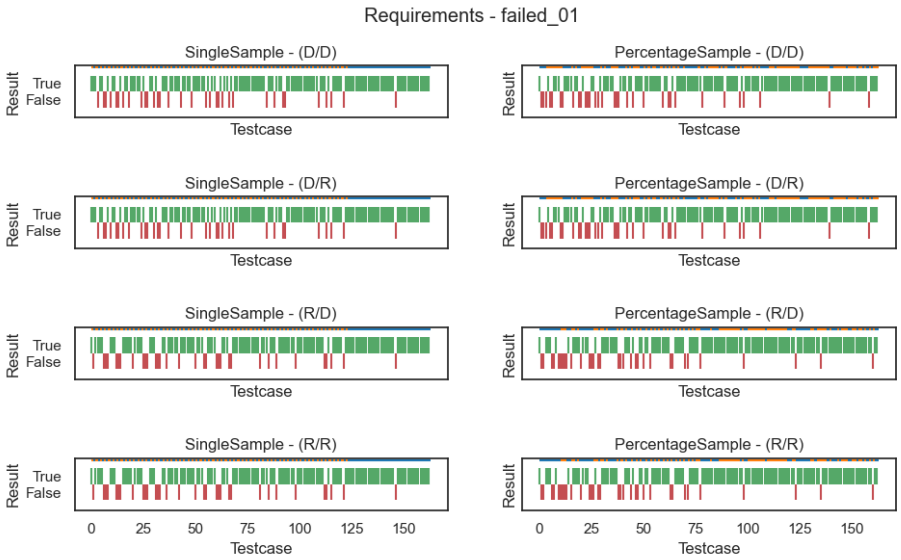


Figure A.7: Results of TCP applied to Model $M_R$ using dataset *failed_01*.

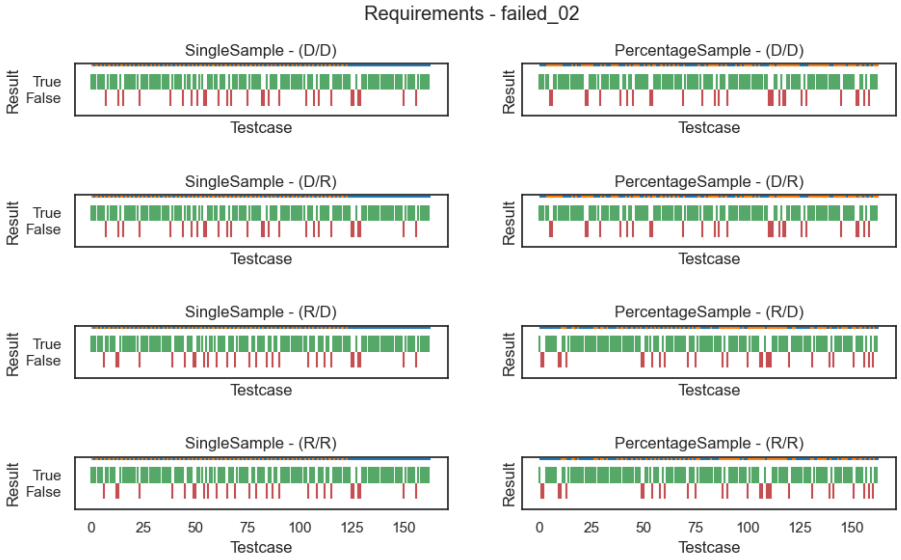


Figure A.8: Results of TCP applied to Model $M_R$ using dataset *failed_02*.

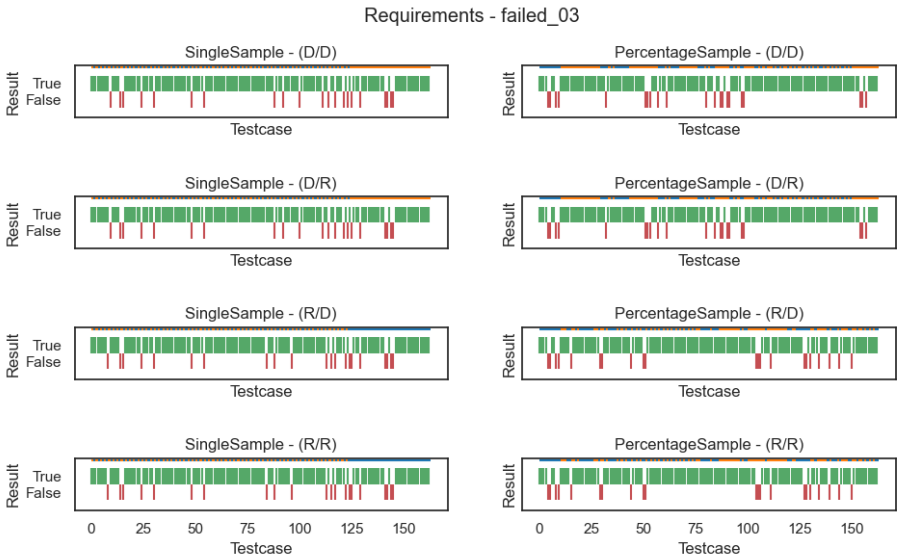


Figure A.9: Results of TCP applied to Model $M_R$ using dataset *failed_03*.

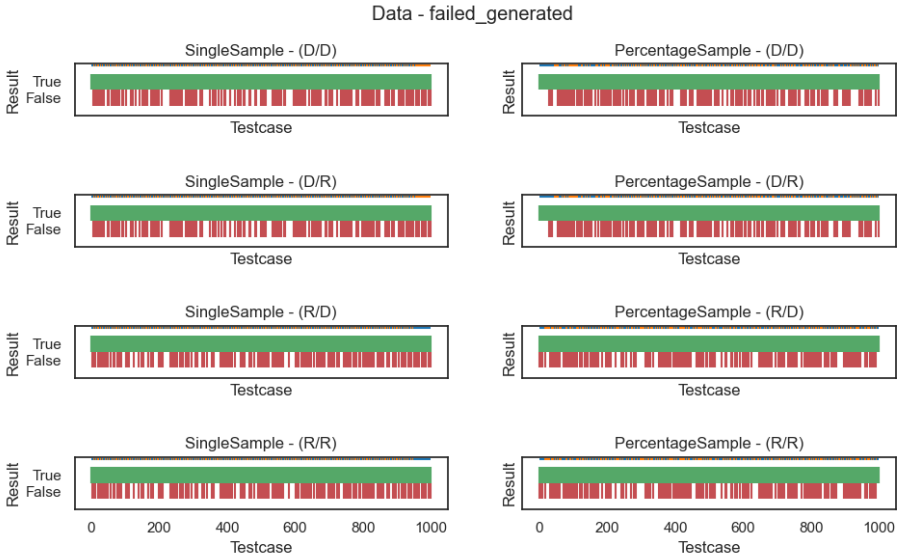


Figure A.10: Results of TCP applied to Model $M_D$ using dataset *failed_generated*.

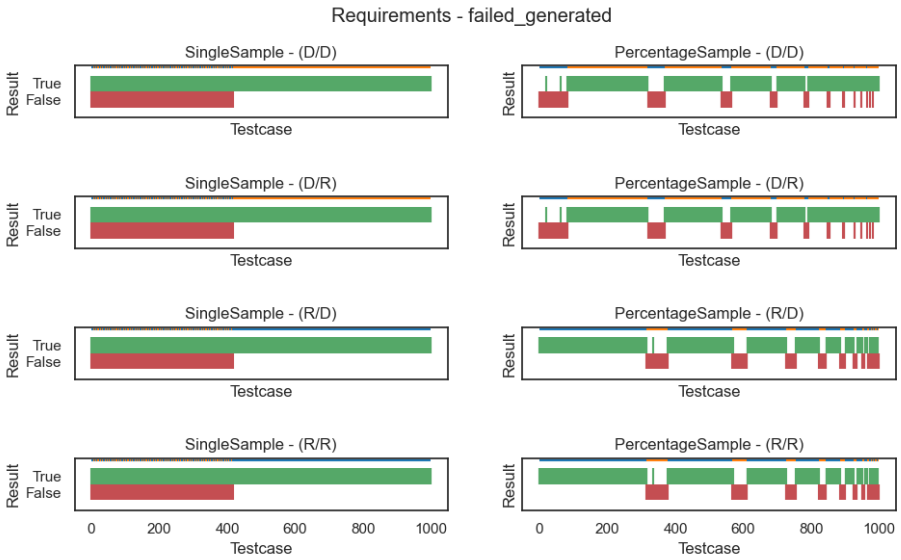


Figure A.11: Results of TCP applied to Model $M_R$ using dataset *failed_generated*.

## A.5 Generated Dataset

A sample of the generated and preprocessed dataset. Represents a failed test case. The data is based on the real data created during the execution of a regression.

**Name**: TC8
**Result**: False
**Duration**: 92
**Data**:

```
init setparam wirebreakcheck true checkQuality true
   checkQuality true checkQuality true checkStatus true
    checkQuality true checkQuality true checkStatus
   true checkQuality true checkQuality true checkStatus
    true checkStatus true checkQuality true checkStatus
    true checkQuality true checkStatus true
   checkQuality true checkQuality true checkQuality
   true checkQuality true checkStatus true checkStatus
   true checkQuality true checkStatus true checkQuality
    true checkQuality true checkQuality true
   checkQuality false result false
```

**Requirements**:

```
pfunction V  pfunction V INT psafety B INT psafety V
   INT psystem B  psystem V INT puser V INT puser B
   psystem V  pfunction B  psafety B  psafety V  puser
   B INT puser V  psystem B INT pfunction B INT
```

**Requirement coverage**: 0.053
**Fault**: 0

# Bibliography


[1] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012 (cit. on p. 5).

[2] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001 (cit. on pp. 5, 11).

[3] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters," in *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, 1996, pp. 226–231 (cit. on p. 8).

[4] R. J. G. B. Campello, D. Moulavi, and J. Sander, "Density-based clustering based on hierarchical density estimates," in *Advances in Knowledge Discovery and Data Mining*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 160–172 (cit. on p. 9).

[5] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proceedings IEEE International Conference on Software Maintenance*, 1999, pp. 179–188 (cit. on pp. 11, 23).

[6] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, 2002 (cit. on p. 11).

[7] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Time-aware test suite prioritization," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, 2006, pp. 1–12 (cit. on p. 11).

[8] P. Tonella, P. Avesani, and A. Susi, "Using the case-based ranking methodology for test case prioritization," in *22nd IEEE International Conference on Software Maintenance*, 2006, pp. 123–133 (cit. on p. 11).

[9] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in *Proceedings of the eighteenth International Symposium on Software Testing and Analysis*, 2009, pp. 201–212 (cit. on p. 12).

[10] R. Carlson, H. Do, and A. Denton, "A clustering approach to improving test case prioritization: An industrial case study," in *27th IEEE International Conference on Software Maintenance*, 2011, pp. 382–391 (cit. on p. 12).

[11] M. J. Arafeen and H. Do, "Test case prioritization using requirements-based clustering," in *IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013, pp. 312–321 (cit. on pp. 12, 16).

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, 2011 (cit. on p. 18).

[13] L. McInnes, J. Healy, N. Saul, and L. Grossberger, "Umap: Uniform manifold approximation and projection," *The Journal of Open Source Software*, vol. 3, no. 29, 2018 (cit. on p. 18).

[14] L. McInnes, J. Healy, and S. Astels, "Hdbscan: Hierarchical density based clustering," *The Journal of Open Source Software*, vol. 2, no. 11, 2017 (cit. on p. 18).