DOI 10.5162/ettc2018/12.4

# Machine Learning-Driven Test Case Prioritization Approaches for Black-Box Software Testing

*Remo Lachmann*
*IAV GmbH, Rockwellstraße 16, Gifhorn, Germany,*
*remo.lachmann@iav.de*

**Abstract**

Regression testing is the task of retesting a software system after changes have occurred, e.g., after a new version has been developed. Usually, only a subset of test cases is executed for a particular version due to restricted resources. This poses the problem of identifying important test cases for testing. Regression testing techniques such as test case prioritization have been introduced to guide the testing process. Existing techniques usually require source code information. However, system testing of complex applications often restricts access to the source code, i.e., they are a black-box. Here, a large set of test cases is manually executed. In previous work, we proposed a test case prioritization technique for system testing using supervised machine learning. We designed our approach to prioritize manually executed test cases, i.e., it analyzes meta-data and natural language artifacts to compute test case priority values. In this paper, we apply further machine learning algorithms and an ensemble learning approach. In addition, we evaluate our approach on three different data sets in total, which all stem from the automotive industry and, thus, represent real life regression testing data sets. We analyze the results of our approach in terms of its failure finding potential. Our findings indicate that black-box testing can be improved using machine learning techniques.

**Key words:** Test Case Prioritization, Black-Box Software Testing, Regression Testing, Machine Learning, System Testing

## 1. Introduction

Modern software systems have to fulfill a large set of requirements due to their complexity and longevity. Thus, in the crucial phase of software testing in a software engineering project, the correspondence of the program behavior to its requirements is assessed. The more complex the system under test, the higher is the testing effort, which takes up to 50% of all project resources in software engineering [13].

While testing is an important task that is part of most professional projects, it still has its limitations. As the testing effort is larger than available testing resources, testing has to be focused on important aspects of the application, which are likely to fail or are of high importance for the overall functionality. However, especially in black-box testing, the identification of important test cases is non-trivial due to the lack of source code availability [26]. White-box test techniques are able to identify changes in the software on code level, which can guide the tester to changes, which should be retested in regression testing.

In contrast, black-box testing is focused on the integrated system [2]. Source code is not available due to various reasons, e.g., contract issues or the usage of precompiled components such as libraries or components developed by 3rd party companies such as suppliers [26]. One prime example of component-based development is the automotive industry, where different companies implement electronic control units, which are later integrated as a whole. Most regression testing techniques focus on source code to select or prioritize test cases [29,16,5]. This reduces their applicability in black-box testing.

To tackle the issue of black-box regression testing, we introduced a regression testing technique based on supervised machine learning in previous work [19]. Our approach prioritizes black-box test cases written in natural language with the aim to emulate test experts and, preferably, find failures as early as possible in the testing process. It uses machine learning (ML) algorithms to find patterns in existing data. Previously, we evaluated our test case prioritization approach using two case studies, one from the automotive industry and one from academia.

In this paper, we diversify and extend our approach using additional ML algorithms to prioritize black-box test cases, i.e., we introduce

the application of neural networks [12], k-nearest neighbor (KNN) [27] and logistic regression [14]. We further extend our concept by combining the output of several ML algorithms to create an *ensemble learner* [27]. Furthermore, we extend our evaluation to a total of three complex industrial systems to make a more general assumption about the effectiveness of our approach. In total, we apply these techniques to three industrial case studies, which provide different features.

In summary, we make the following contributions in this paper:

1. We extend our existing test case prioritization approach by means of additional ML algorithms, which are used in isolation and as an ensemble. We are able to show that our approach is flexible in terms of applied algorithms.
2. We investigate the effectiveness of our technique on three industrial, real-life subject systems. While each system is a software testing project, they are different in nature. Our evaluation results indicate that our approach is indeed applicable to a wide range of projects.

The remainder of this paper is structured as follows: We explain necessary background knowledge in Section 2. Section 3 gives an overview of our general test case prioritization concept. In Section 4, we briefly introduce the machine learning algorithms we apply in this paper to perform a test case prioritization approach. In Section 5, we describe our case evaluation setup and results. We discuss related work in Section 6. We conclude this paper and give insights on future work in Section 7.

## 2. Background

In modern software engineering, testing is one of the most important aspects to ensure software quality [13]. Testing should commence as early as possible and is an important part of each step in the software development process [2]. Especially regression testing is of importance as software development is not stopped after a version has been finished, but rather developed in a continuous fashion, going from one version to the next [20]. Regression testing focuses on the retest of already tested parts of a software system to ensure that changes do not influence previously implemented functionality [21]. To ensure the correct behavior of a system after a change, a full test is necessary, i.e., the execution of each defined test case for each version under test. However, full testing of a software version is not feasible due to restricted resources and complex software systems [7, 29].

To cope with limited testing resources, different regression testing techniques have been developed to reduce the number of test cases to be executed. Most techniques are categorized into *test case prioritization*, *test case selection* and *test case minimization* approaches [29]. All of these techniques are used to guide the focus of the testing efforts. Each approach computes a priority of each individual test case for a particular software version under test based on different criteria, e.g., changed code coverage [16, 29]. While test case selection and minimization aim to reduce the set of executed test cases, e.g., by identifying redundancies, test case prioritization aims to sort the test cases according to their priority.

Prioritization of test cases has one advantage over a selection: it allows continuous testing until resources are exhausted or all test cases are executed, while always focusing on the most important test cases [22]. Test case selection, on the other hand, still requires the full execution of the selected set, which might still be very large. Thus, we will focus on test case prioritization in this paper. In particular, we aim to improve the test case prioritization for black-box testing, where no source code is available. This makes regression testing far more difficult, as traditional techniques analyze source code changes to identify important test cases [7, 16, 29].

To tackle black-box regression testing, we introduced a test case prioritization technique [19] which, is based on ML. ML describes techniques which are able to learn from a given data set to extract information which is then applicable to other data instances [11, 27]. ML techniques are in particular useful for optimization tasks, where a certain optimization goal shall be reached in an efficient manner. ML algorithms are plenty and usually require large data sets to learn from, as this deduction step generates the knowledge required to perform the desired tasks on unknown data.

Different types of ML techniques exist. Two main categories to distinguish algorithms are *supervised* and *unsupervised* ML approaches [27]. Both rely on the notion of *training data*, i.e., data instances provided to extract knowledge. However, in supervised learning these instances contain a *label*, i.e., their corresponding correct output is pre-assigned, e.g., the correct class. For instance, supervised ML techniques can perform classification tasks, e.g., in spam detection [3]. In contrast, unsupervised techniques use unlabeled data. Clustering is one type of an unsupervised approach, where instances are grouped into clusters according to their features [27].

These types of ML approaches have been investigated for decades, and a wide range of specific algorithms have been created to perform specific ML tasks [11, 27]. To apply a specific ML algorithm, the input data is converted into a *feature vector representation.* Each feature is a distinct characteristic, e.g., the sender of a spam mail. The set of all features is a representation of a specific data instance. Based on the feature representation, ML algorithms aim to detect patterns in the data. Examples are similarities between instances, or the prediction of values, e.g., to predict the cost of a house based on its given features and the costs of other houses, which is an example for a regression task.

ML algorithms can also be used together as an *ensemble learning* approach [27]. Here, either different types of algorithms can be combined (*stacking)* or the same type of algorithm can be trained with different input to create a *boosting* approach using weak learners.

This paper focuses on supervised learning as we aim to learn from expert knowledge to predict the importance of new test cases. We also introduce forms of ensemble learning for test case prioritization.

## 3. Concept

The test case prioritization concept of this paper is an extension of our previous work [19]. Fig. 1 illustrates a schematic overview of the main phases of our test case prioritization approach. We give a brief overview of the main steps of the test case prioritization approach in the following.

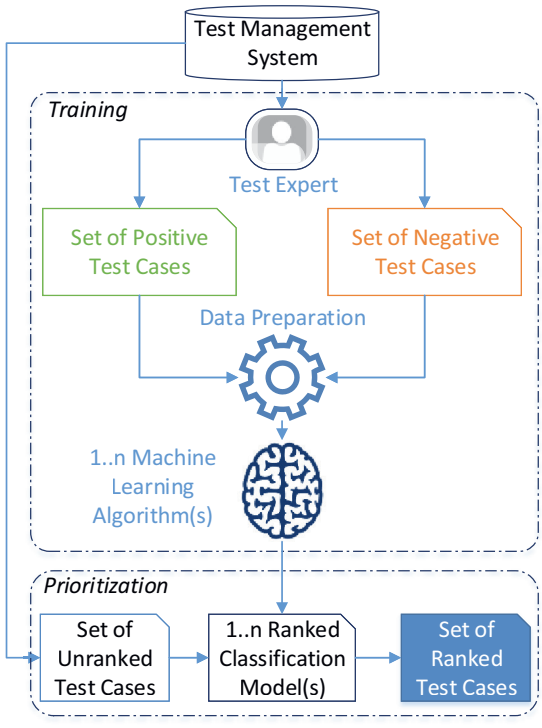


*Fig. 1. Main Concept of the Black-Box Test Case Prioritization using Machine Learning*

As shown in Fig. 1, the data we use to perform a test case prioritization is stored in a data management system. We are not restricted to specific types of data, database or system domain. However, we assume to have access to a defined set of *test cases,* a defined set of *requirements* and a set of *revealed failures.* In theory, each test case is linked to at least one requirement, e.g., how a certain function shall work. Failures, which have been revealed by a test case, should be linked accordingly. However, in practice this traceability is not always given. For our approach, we assume that the authors keep the data up to date and provide traceability between related artifacts.

A test expert has to select a training set for the ML algorithm. In particular, we require the expert to select a set of *positive test cases*, i.e., test cases which are of high importance, used regularly or are for some other reason important for the current version under test. To complement this step, the expert provides a set of *negative test cases*. These are off low importance, e.g., as the particular functionality has not been changed for many versions or is of low risk. Our approach is based on the idea of expert knowledge. Thus, we do not further restrict this step, but let the tester decide about the importance of particular test cases. While this is a manually performed step, we only require the expert to select a subset of test cases.

The training data is used as input for a machine learning algorithm. Our approach is compatible to various ML algorithms, which have to fulfill the following requirements:

- Supervised, as we want to emulate the decisions made by test experts based on two classes: *to test* and *not to test*
- Able to cope with large feature spaces with sparse data
- Result is a ranked classification model, i.e., an input value is provided with an output value, representing its priority

After the training data is selected, we extract features for test cases based on their meta-data, e.g., title, number of linked requirements or execution duration. In addition, we parse their textual description, a novelty among regression testing techniques [19]. Based on the latter, we compute a dictionary of all words contained in the test cases. We prepare the dictionary using natural language processing [6], i.e., tokenization [25], stemming and removal of stop words [28]. Each word represents a feature, which makes the preprocessing useful to reduce the vector space.

We are able to extract the following information for black-box test case prioritization:

- Test case description (natural language)
- Test case age
- Number of linked requirements
- Number of linked defects (history)
- Severity of linked defects
- Test case execution cost (*time*)
- Project-specific features (e.g., market)

Furthermore, we can apply our approach to an arbitrary set of features, which are used by the human expert for test selection. Thus, the set of features, which are considered as input for the machine learning, is selected by an expert before the learning phase commences. This is important for project-specific features.

After the training data is transformed into a feature representation, the ML algorithm computes a ranked classification model. Afterwards, we use this model to prioritize arbitrary, unknown test cases. The result is an ordered list of test cases according to their priority value. The goal is to identify important test cases with a higher likelihood to find failures.

## 4. Applied Algorithms

In this paper, we analyze the effectiveness of four different supervised ML algorithms. In this section, we give a brief overview of the applied algorithms, but refer the reader to additional literature on ML for a more in-depth understanding of their inner workings, which is out of scope for this paper [11, 27].

First, we apply *ranked support vector machines (SVM Rank)* [15] to solve the test case prioritization problem. We used this technique in our previous work [19]. SVM Rank is able to compute a ranked classification model even for large feature vectors and provided good results in previous work for black-box regression testing [19]. Similar to normal SVMs, the algorithm computes a hyperplane in the n-dimensional feature vector space to create maximized margin between two given classes according to their labels.

The second algorithm we apply is *K-Nearest-Neighbor* (KNN) [27]. KNN computes distances between neighbor instances and computes a value according to their labels. The constant *k* defines the number of neighbors, which are considered when computing the class of an unknown instance. For our approach, we set *k* to 5 and use Euclidean distance as these parameters provide the best results.

Third, we apply *logistic regression (Log Reg)* [14]. This technique computes the probability that a given entity belongs to a certain class by fitting a logistic regression curve to the data and performing a maximum likelihood estimation. We use two classes, i.e., to test or not to test. Test cases are ordered according to their probability that they belong to the former category.

The fourth Ml algorithm we apply are *neural networks* [12]. They imitate the human brain and contain *neurons*, which are connected with each other in a layered form. Each neuron might fire given a certain input. We use two hidden layers to solve black-box test case prioritization.

Our approach is able to use all of these algorithms to compute priority (or probability) values for an arbitrary set of test cases, which indicate its importance for the particular system under test. The higher the computed value, the earlier the test case execution.

In addition to an isolated execution of the ML algorithms, we define two more approaches based on the idea of *ensemble learning* [27]. Here, the idea is to combine the results of different algorithms or classifiers to create an even more powerful ML approach. Hence, we introduce *historical* and *combinatorial ensemble learning*. Fig. 2 shows how *n* ML algorithms are used to regression test *m* software versions under test (VUT). The dotted lines represent samples for our ensemble learning concepts.

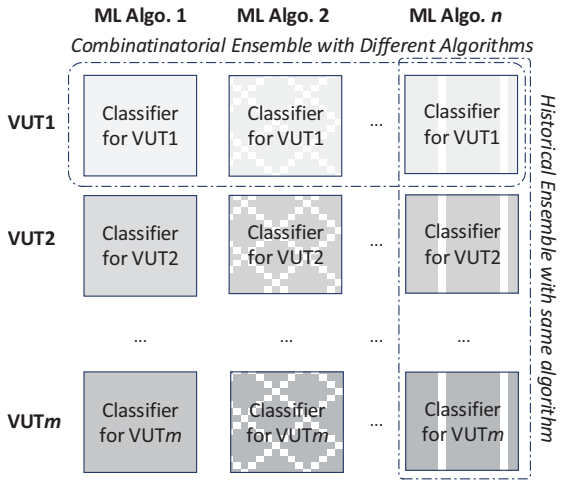


*Fig. 2. Schematic Overview of Historical and Combinatorial Ensemble Approaches*

First, we introduce the idea of *historical ensemble learning* of classifiers as indicated by the vertical dotted line in Fig. 2. Assuming that a software application is tested over the course of several versions. For each version, a new classifier can be trained by testers based on previous findings. Thus, several classifiers exist. Instead of always using the newest classifier, we propose to combine the classifiers of the *n* latest

versions to combine their results and improve the prioritization quality. Similar to the concept of *boosting* [27], we use classifiers of the same type, e.g., only neural networks. Consequently, old classifiers are not dispensed but reused. The impact on the result could be identical or decline with their age, i.e., version number.

Our second ensemble learning approach is to consider a set of classifiers for the same version as indicated by the vertical line in the first row in Fig. 2. Here, we combine the results of (a subset of) *n* different algorithms, e.g., neural networks, SVM, KNN and logistic regression. This combinatorial approach is similar to the concept of *stacking* [27]. When classifying test cases with each of these classifiers, we are able to combine the given priority values for a given test case and compute its average priority value. This provides an overall list for all test cases, which is adjusted according to all classifiers involved. This technique is flexible in terms of what classifiers to apply.

## 5. Evaluation

One of the main aspects of this paper is an analysis of the practical applicability of our test case prioritization approach. We implemented a prototype using the *Dlib.net* ML framework [17] and the *Accord.Net* ML framework [23].

In this paper, we extend our previous evaluation to a total of three industrial subject systems, dropping the academic system. To perform a structured quality assessment, we first describe our research question. Next, the subject systems are explained. Afterwards, we go into detail about the evaluation methodology. Next, we present and discuss the results for all three systems. An assessment of potential threats to validity concludes this section.

### 5.1 Research Questions

For our evaluation, we formulate the following three research questions, which we aim to answer in this paper:

*RQ1:* What is the impact of the test case description features on the quality of the different algorithms?

*RQ2:* Is there one particular ML algorithm, which is the best choice for black-box test case prioritization?

*RQ3:* Is it possible to train the system without the help of an expert to achieve satisfying results?

### 5.2 Subject Systems

In total, we assess our approach using three different subject systems. All three systems stem from the automotive domain. Compared to our previous work, we are able to assess the technique's applicability for real-life data on a larger scale. Due to legal restrictions, we refer to the systems *as System A*, *B* and *C*. Each system describes different types of software testing projects and they involve different authors and stakeholders. We give a basic overview of the size of the three systems in Tab. 1. The table contains the number test cases, which are available for each project, the number of positive (*#Pos TC*) and negative (*#Neg TC*) test cases used for training, the number of failures *(#F*) linked to the training set as well as the vector size, i.e., the number of features extracted. We only use the linked failures in the training data later for evaluation.

*Tab. 1. Subject System Overview*

| ID | #Total TC | #Pos TC | #Neg TC | #F | Vector Size |
|---|---|---|---|---|---|
| **A** | ~1700 | 111 | 115 | 133 | ~1500 |
| **B** | ~2400 | 493 | 278 | 86 | ~3150 |
| **C** | >10.000 | 213 | 133 | 26 | ~6400 |

The three subject systems focus on different software systems and are written by different persons, leading to a high variety on data quantity, quality and content. They even vary in their provided meta-data, i.e., some have user-defined features such as "release market". Our test case prioritization approach is able to handle different data artifacts, i.e., types of meta-data [19]. Thus, we are still able to apply the prioritization to these different projects, even though they do not provide the same features.

### 5.3 Methodology

Our technique aims to provide priority values to perform a guided test case prioritization. Thus, we aim to assess if our technique is indeed able to improve regression testing in terms of effectiveness. In science, this is measured using one particular metric: *Average Percentage of Faults Detected (APFD)* [22]. It computes how fast a set *f* of *m* failures is covered by *n* test cases. It returns a value between 0 and 1, whereas 1 is the theoretical optimum and, thus, the best value. In other words, a higher value indicates that failure revealing test cases are executed first according to the computed priority.

Formally, APFD is defined as follows [22]:

$$APFD = 1 - \frac{\sum_{i=1}^{m} TFi}{n \cdot m} + \frac{1}{2n}$$

We measure APFD for all three different subject systems explained in the previous subsection. In particular, for system B and C, we let a test expert provide the required training data, i.e., a

set of positive and negative test cases. We do not guide the tester in this process. The test experts are only given the information about a desired set size of at least 100 test cases for both sets and that both, positive and negative test sets should be of similar quantity. For *system A*, no expert was available and we had to provide training data on our own, which we further explain in the discussion for *RQ3*.

As we use static data, i.e., are provided with a static set of test cases, requirements and failures, we are not able to detect new failures as test cases are not executed after prioritization. Hence, to compute an APFD value, we have to prepare the data set as follows: We split the set of failures in two subsets according to their age. Old failures are used for training, i.e., the failure content is available as features. New failures are used for testing, i.e., they are used for APFD computation only. The split is done in an uniform fashion, i.e., we split the failures in two sets of similar size where both sets contain about 50% of all failures. This allows us to test the approach based on unknown failures without influencing the trained model.

To perform a more precise analysis of our approaches, we perform a *k-fold cross validation* [27] on our data sets. In particular, we split the set of test cases (comprising both positive and negative test sets) in *k=10* folds. While *k-1* folds are used for training, one fold is used for testing. The testing fold is the one used for APFD computation. Only failures linked to test cases, which are in the testing fold, are considered for APFD computation. Thus, we can make sure that for each repetition of our experiments a new set of “unknown” failures is used for APFD, which reflects the usage of our approach in a real-life scenario. The experiment is repeated *k* times, where the testing fold differs for each repetition to increase confidence in the results.

We compute a random ordering of test cases 100 times for each fold of our cross-validation as comparison. We average the results.

### 5.4 Results and Discussion

**Results for RQ1.** We analyze the performance of our test case prioritization approach using the APFD metric. In particular, we run a k-fold analysis for our three subject systems to assess the effectiveness of our approach compared to a random prioritization.

For the first research question, we investigate how APFD is influenced when using the test case description feature in all combinations with other features compared to those feature combinations, which do not include the test case description.

To give a more detailed sample, we show particular APFD results for all algorithms for system C in Fig.3 and Fig.4. Both figures show boxplots for the different runs of feature-combinations for each technique. The plots show the median value (line in the middle of the box), the average value (only in Fig. 4, cross marker) and upper and lower quartiles (boxes above and below median) as well as the upper and lower boundary of APFD values achieved by each ML algorithm.

In particular, the figures show a boxplot for the random approach, SVM rank (*SVM*), K-nearest neighbor (*KNN*), Logistic Regression (*Log Reg*), Neural Networks (*Neural*) and an ensemble of all four ML algorithms. We use boxplots as we repeated each technique for all different feature combinations, once with test case description (cf. Fig. 3) and without (cf. Fig. 4).

The first thing we notice is that a random ordering produces an APFD of ~0.5. In contrast, most of the ML techniques produce better results. To answer our question, we first analyze the effectiveness when using the test case description feature as seen in Fig. 3. Here, logistic regression performs best of the isolated ML algorithms with an average APFD of 0.69. However, KNN only produces an average APFD of 0.44, which is worse than the random ordering. The best overall result is achieved when combining all four approaches, boosting yields an average APFD of ~0.71, i.e., it achieves the highest failure finding potential for this system.

When analyzing the results displayed in Fig. 4, we notice that feature combinations, which do not include the test case description features do worse than their counterparts. For KNN, the APFD even drops below a random ordering. When the test case description is not available to the ML algorithms, the quality decreases. No algorithm was able to increase its average APFD without using the test case description.

Tab. 2 shows an overview of all average APFD values achieved for all subject systems using the ML algorithms. We split the results for the feature combinations, which included the test case description and for those without. In addition, the average APFD value for each ML algorithm is shown. We mark the best APFD values for each system in bold font. As the results show, the results observed for system C reflect the results we gathered for all subject systems. While a random ordering was not able to produce good results, our applied ML approaches benefit from the usage of the test case description features significantly and are, sometimes, not effective without access to the description (e.g., KNN).

DOI 10.5162/ettc2018/12.4

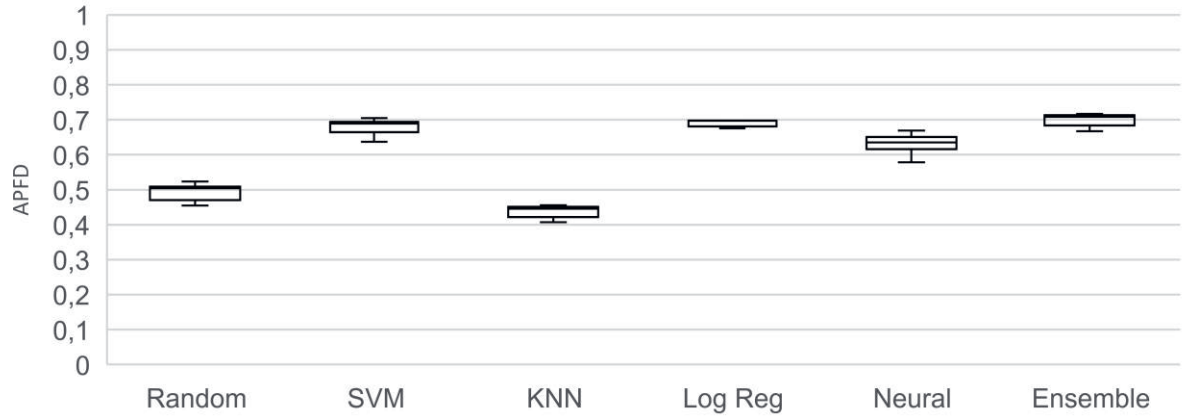


*Fig. 3. APFD Values for Evaluation Repetitions for System C with Description-Related Features*

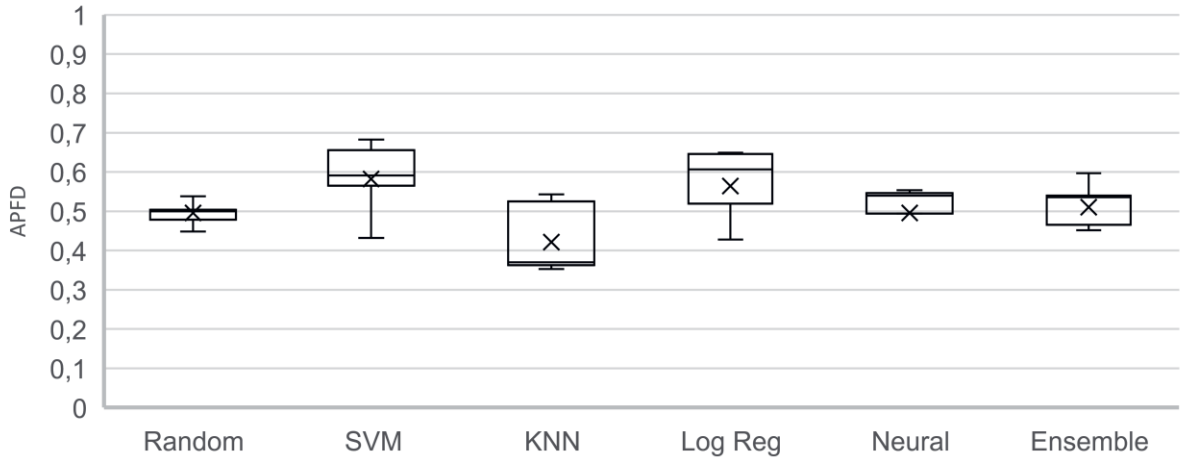


*Fig. 4. APFD Values for Evaluation Repetitions for System C without Description-Related Features*

*Tab. 2. APFD Overview for all three Subject Systems and their Overall Average Value*

| System | Average APFD value with description | | | | |
|---|---|---|---|---|---|
| | *SVM* | *KNN* | *Log Reg* | *Neural* | *Boost.* |
| *A* | 0.68 | 0.69 | **0.75** | 0.7 | 0.72 |
| *B* | 0.64 | 0.52 | **0.66** | 0.56 | 0.57 |
| *C* | 0.68 | 0.44 | 0.69 | 0.63 | **0.7** |
| Avg. | 0.67 | 0.55 | **0.7** | 0.63 | 0.66 |
| | **Average APFD value without description** | | | | |
| *A* | 0.44 | 0.27 | 0.47 | **0.48** | 0.4 |
| *B* | 0.53 | 0.5 | **0.58** | 0.55 | 0.56 |
| *C* | **0.58** | 0.42 | 0.56 | 0.5 | 0.51 |
| Avg. | 0.52 | 0.4 | **0.54** | 0.51 | 0.49 |

In terms of *efficiency,* the most demanding ML algorithm is neural network. For all features selected for our largest system (System *C*), we measure a training time of ~17.3 seconds. KNN is the fastest with about 2.1 seconds. SVM produces a result after ~6.1 seconds and KNN takes 2.3 seconds. The subsequent prioritization is fast for either approach, it only takes milliseconds for each algorithm. These observations are equivalent for all subject systems. The test case description feature has the largest impact on the computational time due to the fact, that each word is a feature increasing the vector space up to 6400 features (cf. Tab. 1).

Overall, logistic regression seems to be the best choice in when investigating effectiveness and efficiency in combination.

**Results for RQ2.** To investigate research question 2, we apply all algorithms and their ensemble to all three subject systems. Tab. 2 shows the average APFD results of all algorithms. We notice that logistic regression outperforms the other applied ML algorithms in System A and B in terms of effectiveness and is

the second best choice in system C. This is the case for all combinations, i.e., with natural language artifacts and without. This shows that the performance of this algorithm is stable and, thus, the logistic regression seems to be a good choice for the test case prioritization approach.

The boosting approach using all four algorithms at once outperforms the logistic regression only in *System C* with an average APFD of 0.7. Thus, the overhead of computing all four approaches seems not worth the effort compared to running logistic regression in isolation. However, it might be worth to investigate a *weighted ensemble approach*, where certain classifiers have a higher impact due to their produced quality [27].

**Results for RQ3.** While we had access to test experts for two subject systems, we had to train *System A* without the help of an expert, which is a restriction to our approach. Therefore, we aim to answer research question 3 using this system.

To provide a meaningful training set without the help of an expert, we select test cases, which have found failures in the past to be of positive impact and, thus, be in the positive test set. The negative set contains only test cases without failures. Furthermore, for this particular project, each test case has a priority and severity value assigned on the 3-level scale by the designers. While these values are sufficient for test case selection, they are too coarse-grained to be used for prioritization. Thus, we use test cases with particular low priority and severity values as negative test cases.

As the results in Tab. 2 indicate, the machine learning was able to produce a sophisticated test case order for *System A*, which outperforms the random ordering. Thus, we validate research question 2. It is possible to train our system without detailed expert knowledge. In particular, the logistic regression algorithm was able to produce good results, with an average APFD value of 0.75.

We notice that it is necessary to provide certain data for training data without expert knowledge. Risk-based data, such as failure severity and impact, is useful to guide the training process, without focusing on experience. This is similar to risk-based testing [10].

### 5.5 Threats to Validity

We aim to mitigate any negative effects, which might influence our evaluation results. While we have developed the tool on our own, which could have caused some faults in the code, we performed intense testing to ensure the correct functionality of our prototype and its parameterization.

To increase the confidence in our results, we use three different case studies. Still, they all stem from the automotive industry, which could influence our findings. However, as different authors and testers maintain these systems and use them in a different context, we argue that our results show that our approach is applicable for different projects to improve a prioritization. Furthermore, we increase the trust in our results using k-fold cross validation.

For our evaluation, we only use a subset of ML algorithms applicable to our problem. Other algorithms might improve the test case prioritization quality even further. To tackle this issue, we used four popular techniques, which already produce desirable results. While further improvement is possible, our algorithm selection already shows the potential of the test case prioritization concept in real-world testing scenarios.

## 6. Related Work

Regression testing has been widely discussed in literature. Yoo and Harman [29] provide a complex survey of minimization, selection and prioritization techniques. Khatibsyarbini et al. [16] show in their survey that the set of test case prioritization publications is still growing in recent years. However, the main focus of most test case prioritization approaches is still a code analysis for priority computation [16].

Machine learning has been used to improve white-box testing in the past. Analyzing the code leads to a wide range of fault prediction approaches, which has been surveyed by Catal [5]. However, white-box approaches require code access, e.g., to analyze modified code snippets for their test relevance. As code access is not available, we have to investigate black-box regression testing approaches in more detail.

In terms of black-box regression testing, our previous work on black-box test case prioritization is most related to the concept presented in this work [19]. We used SVM Rank to compute priority values for natural language test cases. Khatibsyarbini et al. [16] show in their survey, that history and requirements-based approaches, which fall in the category of black-box related techniques, still only share 18% of the total number of publications in novel test case prioritization approaches until 2017. Fazlalizadeh et al. [9] present a greedy technique to prioritize test cases based on system level test data, such as failure history. Engström et al. [7] have extended this approach by using, among others, static priority values. However, they could not find a significant improvement compared to the previous approach. Agarwal et al. [1] use two different ML

techniques as automated oracles in black-box testing. Their findings show that the quality is based on the amount of available data. In contrast to this work, they use program input and output as features. Bhasin and Khanna [4] use neural networks for black-box testing. Here, test cases are defined as module state diagrams. Thus, their approach is model-based in nature and not applicable in a system testing scenario without providing the necessary graphs. De Souza et al. [24] present a multi-objective test case prioritization based on two objectives: minimization of execution costs and requirements coverage. They apply *particle swarm optimization* to achieve their goal. Their results outperform a random approach. In previous work, we defined a multi-objective test case selection approach for black-box testing [18]. We define seven different objectives to be optimized using genetic algorithms. The approach is able to achieve good precision and recall for certain objective combinations. Yoo et al. [30] introduce a clustering approach for test case prioritization. They use dynamic execution traces as input. Human experts prioritize these clusters, which is similar to our idea to incorporate expert knowledge in the process of regression testing.

## 7. Summary and Future Work

**Conclusion.** In this paper, we present different ML-driven approaches for test case prioritization in black-box testing. We extend our previous technique [19] by including other ML algorithms beside SVM Rank [15]. Furthermore, we discuss the idea of ensemble learning-based test case prioritization, i.e., combining the output of different ML algorithms for one version or the output of one algorithm for several versions.

We evaluated our approaches on three different subject systems, which stem from the automotive industry. We ran all four ML algorithms on all three systems and evaluated their effectiveness and efficiency. As baseline, we compared their performance against a randomized ordering. We are able to show that the natural language description of a test case is an important feature for test case prioritization, as it is able to increase the average APFD value for all ML algorithms on all subject systems. We also notice that, given certain meta-data, we are able to train the system without help of an expert.

In total, we are able to state that our test case prioritization approaches are able to outperform a random ordering significantly. The best performance is achieved when using logistic regression [14], which is used for the first time in this paper to solve the test case prioritization problem based on natural language artifacts.

**Future Work.** In our evaluation, we noticed a partial lack of traceability between artifacts, e.g. the link between failures and test cases is not always provided. The main reason for this issue is explorative testing, where testers do not strictly follow a protocol but rather test in a use-case driven environment. An example for this are test drives in the automotive industry, where testing is performed in an ad-hoc fashion.

The issue of missing traceability is reducing data quality and, therefore, the potential of our machine learning-driven test case prioritization approaches. Thus, we are investigating techniques, which are able to improve traceability in a semi-automatic fashion. By analyzing failures, we want to create automatic links to suitable test cases, which might have produced this failure. This improves data quality, test case coverage analysis and the potential for automatic test case prioritization. Furthermore, we want to investigate the potential of our historical ensemble approach to prioritize test cases. This requires a long-term evaluation setup, where different versions are tested in a live testing environment.

## Acknowledgements

The author thanks Antony Beno Louison Jayapathy for fruitful discussions and his help with the realization of the presented concept.

## References

[1] D. Agarwal, D. E. Tamir, M. Last and A. Kandel. A Comparative Study of Artificial Neural Networks and Info-Fuzzy Networks as Automated Oracles in Software Testing," in IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans, vol. 42, no. 5, pp. 1183-1193, 2012: doi: 10.1109/TSMCA.2012.2183590

[2] P. Ammann and J. Offutt. Introduction to Software Testing (1 ed.). Cambridge University Press, New York, NY, USA, 2008. ISBN: 978-0521880381

[3] F. Benevenuto , G. Magno , T. Rodrigues , V. Almeida. Detecting Spammers on Twitter. Proceedings - Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS), 2010. doi:10.1.1.297.5340

[4] H. Bhasin and E. Khanna. Neural network based black box testing. SIGSOFT Softw. Eng. Notes 39 - 2, 2014, pp. 1-6. doi: 10.1145/2579281.2579292

[5] C. Catal, Software fault prediction: A literature review and current trends, Expert Systems with Applications, Volume 38, Issue 4, 2011, pp. 4626-4636, doi:10.1016/j.eswa.2010.10.024

[6] G. G. Chowdhury . Natural language processing. Annual review of information science and technology, 37(1), 2003, pp. 51-89. doi: 10.1002/aris.1440370103

[7] E. Engström, P. Runeson and A. Ljung, "Improving Regression Testing Transparency and Efficiency with History-Based Prioritization -- An Industrial Case Study," 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, 2011, pp. 367-376. doi: 10.1109/ICST.2011.27

[8] G. Erdogan, Y. Li , R.K. Runde et al. Int. Journal on Software Tools Technology Transfer, 2014, 16: 627. doi: 10.1007/s10009-014-0330-5

[9] Y. Fazlalizadeh, A. Khalilian, M. A. Azgomi and S. Parsa, "Prioritizing test cases for resource constraint environments using historical test case performance data," 2009 2nd IEEE International Conference on Computer Science and Information Technology, 2009, pp. 190-195. doi: 10.1109/ICCSIT.2009.5234968

[10] M. Felderer and I. Schieferdecker. 2014. A taxonomy of risk-based testing. Int. J. Softw. Tools Technol. Transf. 16 - 5, 2014, pp. 559-568. doi:10.1007/s10009-014-0332-3

[11] A. Géron. Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems. O'Reilly Media, Inc., 2017. ISBN: 978-1491962299

[12] I. Goodfellow, Y.Bengio, A. Courville, “Deep learning”. Vol. 1. Cambridge: MIT press, 2016. ISBN: 0262035618

[13] M. J. Harrold, Testing: a roadmap. In Proceedings of the Conference on The Future of Software Engineering (ICSE), 2000, ACM, pp. 61-72. doi: 10.1145/336512.336532

[14] D. W. Hosmer Jr., S. Lemeshow, and R. X. Sturdivant. Applied logistic regression. Vol. 398. John Wiley & Sons, 2013. doi:10.1002/9781118548387

[15] T. Joachims, “Optimizing search engines using clickthrough data”. In Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '02). ACM, pp. 133-142. doi: 10.1145/775047.775067

[16] M. Khatibsyarbini, M. Adham Isa, D. N.A. Jawawi, R. Tumeng, Test case prioritization approaches in regression testing: A systematic literature review, Information and Software Technology, Vol. 93, 2018, pp. 74-93. doi: 10.1016/j.infsof.2017.08.014

[17] D. E. King. 2009. Dlib-ml: A Machine Learning Toolkit. J. Mach. Learn. Res. 10 (December 2009), pp. 1755-1758. doi: 10.1145/1577069.1755843

[18] R. Lachmann, M. Felderer, M. Nieke, S. Schulze, C. Seidl, and I. Schaefer. Multi-objective black-box test case selection for system testing. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '17). ACM, 2017 pp. 1311-1318. doi: 10.1145/3071178.3071189

[19] R. Lachmann, S. Schulze, M. Nieke, C. Seidl and I. Schaefer, "System-Level Test Case Prioritization Using Machine Learning," 2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA), 2016, pp. 361-368. doi: 10.1109/ICMLA.2016.0065

[20] M. M. Lehman, "Programs, life cycles, and laws of software evolution," in Proceedings of the IEEE, vol. 68, no. 9, 1980, pp. 1060-1076. doi: 10.1109/PROC.1980.11805

[21] H. K. N. Leung and L. White, "Insights into regression testing (software testing)," Proceedings. Conference on Software Maintenance, 1989, pp. 60-69. doi: 10.1109/ICSM.1989.65194

[22] G. Rothermel, R. H. Untch, Chengyun Chu and M. J. Harrold, "Prioritizing test cases for regression testing," in IEEE Transactions on Software Engineering, vol. 27, no. 10, pp. 929-948, 2001. doi: 10.1109/32.962562

[23] C. R. Souza, "The Accord.NET Framework," http://accord-framework.net . São Carlos, Brazil. 2014.

[24] S. de Souza, Luciano & Miranda, Péricles & Prudêncio, Ricardo & Barros, Flávia. A Multi-Objective Particle Swarm Optimization for Test Case Selection Based on Functional Requirements Coverage and Execution Effort. Proceedings - International Conference on Tools with Artificial Intelligence, ICTAI. pp. 245-252, 2011. doi: 10.1109/ICTAI.2011.45.

[25] J. J. Webster and C. Kit. Tokenization as the initial phase in NLP. In Proceedings of the 14th conference on Computational linguistics - Volume 4 (COLING), Vol. 4., 1992, pp. 1106-1110. doi: 10.3115/992424.992434

[26] E. J. Weyuker, "Testing component-based software: a cautionary tale," in IEEE Software, vol. 15, no. 5, pp. 54-59, 1998. doi: 10.1109/52.714817

[27] I. H. Witten, E. Frank, and M. A. Hall. 2017. Data Mining: Practical Machine Learning Tools and Techniques (4th ed.). Morgan Kaufmann Publishers Inc. ISBN: 978-0-12-804291-5

[28] Z. Yao and C. Ze-wen, "Research on the Construction and Filter Method of Stop-word List in Text Preprocessing," 2011 Fourth International Conference on Intelligent Computation Technology and Automation, Shenzhen, Guangdong, 2011, pp. 217-221. doi: 10.1109/ICICTA.2011.64

[29] S. Yoo and M. Harman, Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verif. Reliab., 22: pp. 67-120, 2012. doi:10.1002/stvr.430

[30] 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 (ISSTA). ACM, pp. 201-212, 2009. doi: 10.1145/1572272.1572296