System-Level Test Case Prioritization Using Machine Learning

Remo Lachmann, Manuel Nieke, Christoph Seidl, Ina Schaefer
Technische Universität Braunschweig, Germany
{r.lachmann, manuel.nieke, c.seidl, i.schaefer}@tu-bs.de

Sandro Schulze
Otto-von-Guericke Universität Magdeburg, Germany
sanschul@iti.cs.uni-magdeburg.de

***Abstract*—Regression testing is the common task of retesting software that has been changed or extended (e.g., by new features) during software evolution. As retesting the whole program is not feasible with reasonable time and cost, usually only a subset of all test cases is executed for regression testing, e.g., by executing test cases according to test case prioritization. Although a vast amount of methods for test case prioritization exist, they mostly require access to source code (i.e., white-box). However, in industrial practice, system-level testing is an important task that usually grants no access to source code (i.e., black-box). Hence, for an effective regression testing process, other information has to be employed. In this paper, we introduce a novel technique for test case prioritization for manual system-level regression testing based on supervised machine learning. Our approach considers black-box meta-data, such as test case history, as well as natural language test case descriptions for prioritization. We use the machine learning algorithm SVM Rank to evaluate our approach by means of two subject systems and measure the prioritization quality. Our results imply that our technique improves the failure detection rate significantly compared to a random order. In addition, we are able to outperform a test case order given by a test expert. Moreover, using natural language descriptions improves the failure finding rate.**

***Index Terms*—System-Level Testing, Black-Box Testing, Test Case Prioritization, Supervised Machine Learning**

I. Introduction

Modern software systems are usually very complex, thus, testing plays a pivotal role in the entire development process. Moreover, software undergoes a continuous evolution process either to fix known failures or to introduce new features. Hence, for every version to be released, *regression testing* is a fundamental task, as it ensures adherence to the software's specification. Testing requires at least 50% of resources allocated to a software project [16]. Unfortunately, it is too expensive to test a software version exhaustively due to limited resources available for a usually very high number of test cases. In contrast, especially safety-critical software systems require a comprehensive testing process, for instance, to fulfill standards, such as ISO 26262 [1]. Test case prioritization allows to focus on the most important test cases [34]. However, current prioritization techniques mostly consider *white-box testing*, that is, access to source code is available [3].

In contrast, many complex software systems are developed in a component-based fashion, where different components come from independent suppliers or teams within a company [10]. It is common in system integration, that access to component internals is restricted, i.e., the source code of the components is unknown to the tester. Hence, *black-box* testing is applied, which is based on a system specification containing requirements in natural language. In addition, test automation might not be available. Test cases are defined in natural language and executed manually. In such a setting, regression testing becomes difficult as test case prioritization techniques are usually applied with access to source code [13]. There don't exist any techniques to our knowledge, which are able to exploit natural language test case descriptions to improve regression testing.

To fill this gap, we propose a novel technique that allows for automatic test case prioritization in system-level testing of test cases written in natural language. It aims to emulate test expert knowledge and to reuse it for an automatic prioritization of test cases. We employ supervised machine learning (ML) and natural language processing (NLP) to rank test cases. As a result, the failure detection rate of testing is improved by finding failures earlier. This increases testing effectiveness compared to random and manual prioritization, especially, as prioritizing test cases manually a priori is infeasible and done in an ad-hoc fashion.

In this paper, we make the following contributions:

- A concept for test case prioritization in system-level regression testing based on black-box meta-data and natural language test case descriptions, improving the prioritization of test cases compared to random and manual ordering.
- A realization of our concept using a ranked classification ML algorithm, the *ranked support vector machine* (SVM RANK), obtaining a ranked classification according to the priority of test cases.
- An evaluation of our technique with two subject systems, one from academia and one based on industrial test data. Our results show that our method considerably improves the fault revealing rate compared to a random approach and outperforms manual prioritization of test cases in terms of necessary time and fault detection rate.

II. Background

A. Black-Box Regression Testing

Software is often developed by several parties, such as suppliers, and assembled and tested as a whole by an OEM against its specification [17]. A prominent example for *component-based* development is the automotive industry, where this

approach is commonly applied to software development for electronic control units. If no automation is available in black-box testing, the corresponding test cases are executed manually and, thus, test cases descriptions are defined in natural language artifacts containing different test steps to be executed by a tester. In practice, test data is stored in test management systems or similar applications and updated continuously throughout a project's life cycle [10]. Due to the high complexity of testing, test sets are created manually in system-level testing and, thus, come with high costs.

Test set creation is a recurring task in *regression testing*, which is the process of retesting a particular version of an application after it has been modified [26]. In this paper, we particularly focus on *test case prioritization*, which aims at ordering test cases according to a *priority* value, in order to detect failures or to execute test cases covering risky components early [34]. As a result, testers can execute ordered test cases until either resources or time are exhausted, ensuring to execute the most important test cases. In this paper, we focus on prioritizing test cases according to the decisions made by a test expert imitating the expert's behavior using ML.

B. Applied Machine Learning Techniques

Classic *Support Vector Machines* (SVM) try to find a *hyperplane* in a high-dimensional vector space, separating classes of data. Ideally, the margin between data points of different classes is maximized. Data points with minimum distance to the hyperplane are *support vectors* [9]. New data points can be assigned to one of the resulting classes using the resulting hyperplane function.

As a binary data split does not support test case prioritization, we apply ranked classification techniques. For example, relevance vector machines (RVM), which use similar functions as SVMs, provide a probabilistic classification for each data point. The classification indicates the probability that a data point belongs to a certain class [29], [4]. Compared to a binary classification, each data point is assigned a value between 0 and 1, which constitutes the probability for belonging to one of the classes. As a result, data instances can be ordered by their probability values.

For this work, we apply the *ranked support vector machine* (SVM RANK), which has been introduced by Joachims [19] to optimize the retrieval quality of search engines for prioritization. SVM RANK returns a ranked classification function for supervised training input and is capable to handle input vectors of large size [19]. We also experimented with RVM [29], but runs even on smaller data sets indicated that this technique is not applicable as it struggles with large feature vector sizes.

For evaluation, we apply *k-fold cross validation* [31]. Here, the training set is split in k parts, where each part is used once as validation set, whereas the other $k-1$ parts are used as input for the training. The training is repeated k times, such that each fold is used for validation once.

III. PRIORITIZATION METHODOLOGY

In Figure 1, we show our test case prioritization methodology, consisting of five phases explained in the following.

A. Training Data Selection

As we apply supervised machine learning to rank test cases, we require that a set of test cases are labeled as either *important* or *unimportant* by an expert as training data. Using labels defined by experts supports the goal of our approach is to emulate testers decisions and knowledge. The training data quality has a significant impact on the prioritization result.

Certain aspects should be kept in mind when selecting training data [15]. The number of class entities has to be balanced, i.e., the number of test cases ideally is similar for both classes. Furthermore, the number of features is significant, as high dimensionality leads to potential large computation times and reduction of prediction performance [15]. In our case, the dictionary size leads to a considerable increase in the number of features. However, SVM Rank was able to cope with more than $3,000$ different words in real-world dictionaries.

Once the training set has been selected, a feature vector is generated for every test case based on its attributes.

B. Dictionary Creation and Meta-Data Collection

We assume that test case descriptions (TCD) are written in natural language, comprising different steps to be executed by the tester and the expected result. To make TCDs processable for machine learning, we create a *dictionary* containing all words that occur within all test cases. Each word represents a feature for ML. The corresponding number of word occurrences in a test case is used as value. This knowledge is used to detect patterns contained in selected test case descriptions, e.g., whether test cases have been selected for a similar topic. To reduce redundancy and ambiguity between words, we apply the NLP techniques *tokenization* [30], *filtration* [33] and *stemming* [24]. Words are reduced to their stem and stop words and punctuation marks are removed. To reduce the feature space, we only compute the dictionary based on test cases selected as training data, as they lead to our ranked classification model. The dictionary is used as model representing a test case's characteristics to later compute a ranked classification model. To prevent overfitting, the number of features (i.e., words) might have to be reduced to a certain glossary, if not enough training data samples are available.

Besides the *test case description (TCD)*, our technique relies on further meta-data attributes provided by the test management system, which serve as input for the learning algorithm. Overall, we collect the following data:

Requirements Coverage (RC): Each test case corresponds to at least one requirement. Hence, requirements can be used as input for ML, for instance, to check which requirements are covered by a particular set of test cases. Each requirement is represented as feature for ML. The feature value for a requirement is either 0, if the requirement is not covered by the test case or 1 otherwise.

Revealed Failures: All failures which have been found by a test case are tracked. Hence, the *failure count (FC)* of a test case is used as another attribute for learning. This is particularly interesting for failure retests to ensure old failures have been fixed. Moreover, for every reported failure, the

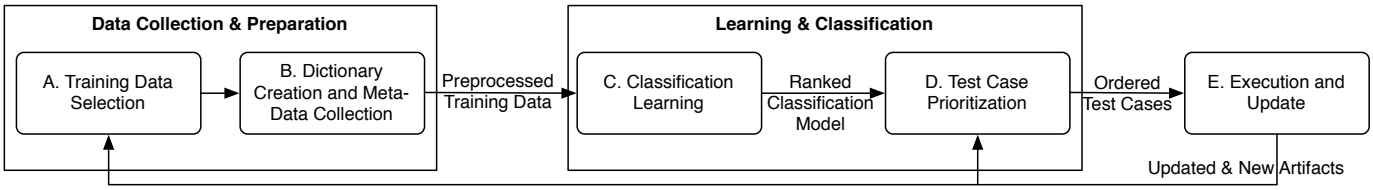

Fig. 1: Methodology of the Test Case Prioritization Technique

failure age (FA) (i.e., date when it was found) and the assigned *failure priority (FP)* are recorded. The FA attribute is of interest if only new failures shall be considered for a retest. We define FP on a range from 1 to 3, where 1 indicates a high severity failure that absolutely has to be fixed before release, 2 are test cases of normal importance and 3 are cosmetic failures. Each attribute is a feature of ML, the value FC representing the sum of failures revealed by test case, FA their average age and FP the sum of the failure priorities, respectively.

Test Execution Cost (EC): We keep track of the time it takes to manually execute a test case (i.e., its cost), e.g., by a test management tool. The average cost of a test cases is computed as average of all costs gathered over the set of all runs for this test case. This attribute can also be used to identify test cases which have not been run at all. The EC criterion has to be used with caution as the recorded time can be very subjective.

C. Classification Learning

The main goal of our technique is to rank test cases based on a given set of training data so that it corresponds to the intuition of a human tester creating a test set. To this end, we employ SVM RANK (cf. Sec. II-B) to learn a ranked classification model (classifier) based on given supervised training data. Due to the flexibility of our approach, other ranked classification algorithms can be applied as well.

The goal of the prioritization is to reduce testing effort in two ways. First, we reduce the effort for creating suitable test sets because a tester only has to classify a subset of test cases used as training data. The resulting classifiers are reusable for different test cases. Second, we increase the testing effectiveness as failures are found earlier compared to a random or manual prioritization. Hence, test execution can be stopped earlier if resources are exhausted.

Due to software evolution, classification models get outdated and, thus, new classifiers have to be learned. Based on our experience, the interval between learning of new classifiers is project-specific, but not required as often as manual selection or prioritization of test cases.

D. Test Case Prioritization

Once a ranked classification model has been learned, it is applicable for prioritization of other test cases. Selecting test cases as input for our technique defines the *scope* of the prioritization, i.e., which test cases shall be prioritized. In current practice, a manual prioritization of several hundred test cases is infeasible. Our approach improves this by automatically prioritizing large number of test cases in reasonable time.

After selecting the test cases, subject to prioritization, they are transformed into a vector representation. Next, these vectors are fed into the classifier to return a probability value. Afterwards, we order test cases according to their computed probability value in descending order and combine them to a new *test set*, which is uploaded to the test management system. For a better understanding of the ordering, we assign the probability values to test cases, which also allows for defining a test end criterion. In particular, a *threshold* can be defined specifying a lower bound for the probability value up to which test cases should be executed.

E. Execution and Update

A learned ranked classification model (classifier) is reusable as often as desired. The normal use case is that a regression test is to be performed after a software update, i.e., components have been updated by a third party supplier and the system has to be tested before release. In the long run, continuous evolution of the project may induce repeating the classification learning step (cf. Section III-C). In particular, the following evolutionary changes may require a new classifier:

New Failures Revealed: If a failure has been revealed, the test case should be retested after the fix. Additionally, the failure could imply that a certain part (e.g., a module) of the software is erroneous.

New/Changed Requirements: Requirements are regularly updated or added, especially for new major releases, which contain new features. This indicates important changes, which require a new classifier.

New/Changed Test Cases: New test cases can be used as input for prioritization, depending on the current project status. In addition, a new classifier, if many new test cases have been added or test cases have been changed.

New Functionality: Within the project's life cycle, features are developed continuously, i.e., different features are available at different times for testing. Consequently, training data should be updated to learn a new classifier, encompassing new functionality. This is done by extending positive test sets to cover new requirements and updating negative test cases. Creating new training data is less tedious than manually prioritizing and selecting test cases for each test set.

The decision whether to learn a new classifier has to be made by test experts, depending on the success rate of previous classifiers and the degree of changes within the project. In any case, previously learned classifiers are not lost as we store them persistently and, thus, classifiers are reusable at any time.

IV. Evaluation

We evaluate our technique to examine how it improves the failure revealing rate. We present the research questions, subject systems used, evaluation methodology and the results.

A. Objectives

We inspect the following two research questions:

RQ1: *How good is the quality of the prioritization results of our proposed technique (1) compared to a random prioritization and (2) compared to the manual live execution of test cases as it is currently applied?* This is our main objective as we want to achieve a higher effectiveness, i.e., we want to find failures earlier. Thus, we compare our approach to a naive, random approach, which we consider as a baseline. Moreover, we compare our approach with a prioritization by test experts on real-world data.

RQ2: *How does the test case description, used for learning, affect the results and runtime of the ML technique compared to the other attributes?* With our technique, we overcome existing black-box approaches by including also the descriptions of test cases. We investigate whether this improves prioritization compared to more basic metadata such as test history or execution time.

B. Subject Systems

We evaluate our technique using two subject systems, for which we have *natural language test case descriptions* (TCD).

Subject System 1: Body Comfort System. The first subject system constitutes an automotive Body Comfort System (BCS) [21], e.g., an automatic power window and alarm system. BCS is an academic project, originating from a real product and thus, containing all data necessary to apply our prioritization technique. In particular, 97 requirements and 128 test cases are defined. Both, requirements and test cases, have been written by students instead of professional testers. The dictionary encompasses 180 words after preprocessing.

For training, we split the 128 test cases equally into 64 positively and 64 negatively labeled test cases, thus, preventing class imbalance (cf. Section III-A). We manually assigned these labels according to a the priorities of the different components of the system, e.g., the alarm system has a higher importance than the exterior mirror heating. A total of 7 failures has been seeded. Unfortunately, no execution costs are available, as the system is not available for testing.

Subject System 2: Automotive Industry Data. As second system, we use a real-world system provided by an industrial partner from the automotive domain. The system is developed continuously with requirements, test cases, and failures described in natural language. In particular, the available data comprises more than $10,000$ requirements, $10,000$ test cases, and $1,000$ reported failures. Artifacts are provided within HP Quality Center(QC)[1], a test management system.

[1]Part of HP Application Lifecycle Management, website: http://www8.hp.com/us/en/software-solutions/application-lifecycle-management.html

Currently, manual prioritization is costly . The manual test process is as follows: the test engineer groups test cases that should be executed according to different scenarios and performs an (implicit) ordering of these groups according to his domain knowledge about which parts should be tested.

Due to the large number of test cases and missing expert knowledge about the system, it was not feasible for us to assign labels to all test cases. We use a subset 354 positively and 291 negatively labeled test cases. The labeling has been supported by test experts. Moreover, a total of 34 failures are known for the (positive) training data set. The computed dictionary consists of about $3,500$ different preprocessed words.

C. Evaluation Methodology

Next, we explain our evaluation methodology regarding data preparation, measurements, and implementation.

Data Preparation. We performed multiple runs of our prioritization technique for each system, selecting six different attributes and their combinations as feature input: test case description (TCD), requirements coverage (RC), failure count (FC), failure age (FA), failure priority (FP) and execution costs (EC). This leads to a total of $2^6 - 1 = 63$ runs for all attribute combinations. Due to seeded failures, the EC attribute is unknown for BCS and, thus, only $2^5 - 1 = 31$ runs are performed. For each of these runs, we perform *k-fold cross validation* [31], which is applicable as test cases do not depend on each other. For BCS, we use $k = 5$ folds, due to the small data set, and for industrial data, we use $k = 10$ folds. To make the results more comparable, we use the same folds for each run using different combinations of attributes.

To cope with high dimensionality of EC and FA, we change the FA feature from the concrete number of days to a 9-point scale, based on the time passed since their detection, leading to the following values: "*today*" (1), "*last week*" (2), "*two weeks*" (3), "*last month*" (4), last "*three months*" (5), within "*six months*" (6) and "*last year*" (7) or older than that (8). If no failures have been found, a value of 0 is given. Similarly, we introduce a 4-point scale for EC, based on the time needed for test execution: 1 = *short* (< 30 seconds), 2=*medium* (< 5 minutes) and 3 = *long* (> 5 minutes). We apply a value of 0 if a test case has never been executed.

Execution and Measuring. We applied our prioritization as described in Section III. As measurement for effectiveness of the prioritization, we use the commonly used *Average Percentage of Faults Detected* (APFD) [27] metric. We use APFD to measure which failure f was found at which position TF_i in the ordered test set consisting of $i = 1 \dots n$ test cases. APFD computes values between 0 and 1, with higher values indicating that failures are found earlier. Different APFD metrics exist, e.g., by measuring test case costs [11]. We use the position a of test case in an ordering as our test cases are executed manually and the position of a test case, corresponding to a failure found, is a good indicator of the prioritization quality as testers have an idea of how many test

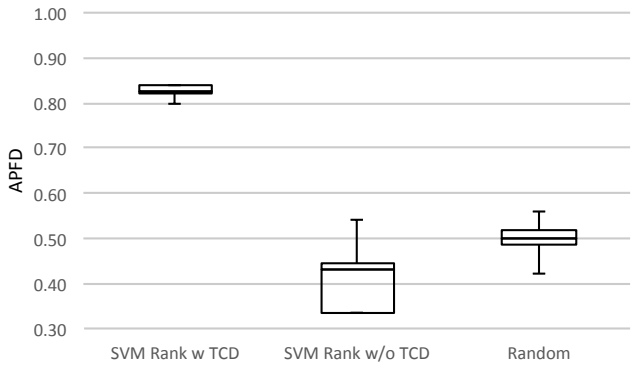

Fig. 2: APFD Results Overview for BCS

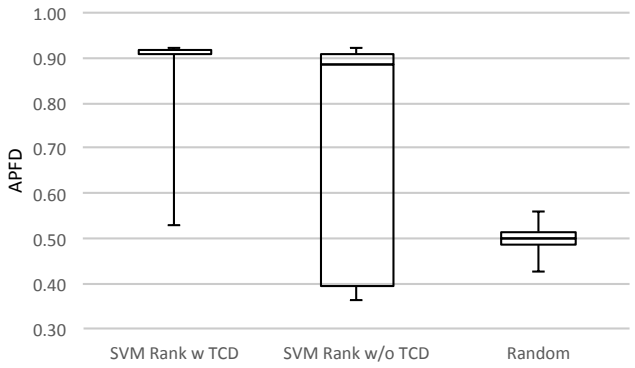

Fig. 3: APFD Results Overview for Industry Data

cases they are able to execute in a certain time frame. APFD is defined for *n* test cases and *m* failures as follows [27]:

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

To reduce data pollution, we consider only failures connected to positively labeled test cases for APFD computation. We perform additional preparations for the failure measurement in case a run relies on failure-related attributes, i.e., FC, FA or FP. We split the failures by their average failure age using the "older" failure set for training. In contrast, for computing APFD, we only use the second or "newer" failure set. This way, we emulate a failure finding process based on *old* failures to find newer, *unknown* failures. Otherwise, the failure distribution may bias the evaluation as failures would be used for both building the classification model and computing the APFD. We apply the aforementioned process only in case when failure-related attributes are used for learning. Consequently, three failures for BCS are used to measure APFD and five failures are used for training. APFD for industrial data is computed using 16 out of 34 failures. We measure the APFD for each fold for each run. Finally, we compute the average APFD per run (i.e., for each attribute combination), used to evaluate the impact of certain attributes on the prioritization. Beyond APFD, we measure the time it takes to train the ranked classification model to evaluate the applicability of our approach in practice. Finally, we normalize the random ordering results by running the randomization 100 times for each iteration of cross validation, i.e., we create 100 prioritized test sets for *k* repetitions of the k-fold cross validation and compute the average values.

Implementation. We use QC to store data of our subject systems, i.e., providing traceability between all requirements, test cases and failures. We implemented our proposed technique in a component that connects to QC to retrieve and send data and applies the machine learning algorithm to prioritize test cases. For the implementation of the ML algorithm, we use the *dlib* [20] machine learning library.

D. Results

Next, we present and discuss our case study results, grouped by the research questions we defined in Section IV-A.

RQ1. *Body Comfort System:* In Fig. 2, we show the results for the APFD metric by means of two boxplots for SVM Rank and one for random prioritization. Each boxplot represents the average APFD values for each run performed for the different combinations of attributes, split into combinations with and without the TCD attribute.

Our data reveals that our technique considerably outweighs random prioritization. In particular, it achieves a combined median APFD value of 0.7 for both boxplots with a peak value greater than 0.8. This is higher than the average APFD of about 0.5, which the random algorithm achieves.

Industrial Data. We show the results for our industrial data in Fig. 3. Again, our technique outweighs random prioritization having a overall median APFD value of almost 0.9 in general for industrial data. This indicates that the technique scales very well with larger dictionaries as well as larger quantities of input data.

The APFD results indicate that our technique provides a very competitive prioritization of test cases. To strengthen this observation, we also performed an additional comparison of our technique to the currently applied manual test process of test experts of an industrial partner. The comparison was performed in a live test run on a current version of an industrial automotive SUT. Hence, we measured the detection rate of new failures. For fair comparison, we also let the test expert select the training data. These training sets led to two classification models, referred to as *Classifier A* and *Classifier B*. The first training set was selected without any previous knowledge about the technique. Classifier B has been created after Classifier A had been used for prioritization. In particular, the test expert selected 237 positive and 158 negative test cases as a first training data set (for *Classifier A*), and 255 positive and 155 negative, as second set (for *Classifier B*).

To gain insight on our performance, we selected a total of 155 test cases, which will be used to measure the APFD of our technique, i.e., these test cases are prioritized using

both learned classifiers. Only a small subset of the test cases have been used for training. The test case quantity is rather low to ensure that all test cases are executed to gain knowledge on revealed failures used for APFD computation. Afterwards, we let the test expert execute the test cases in an ordering according to his knowledge and personal decisions (as described in Sec. IV-B).

Fig. 4 shows the resulting graphs of the failure detection rate of the manual and automatic prioritization. Our data reveal that the results of both classifiers outperform the manual ordering considerably. The manual approach (yellow cross-dotted line) achieves an APFD value of 0.53, while the automatic prioritization leads to an APFD value of 0.60 and 0.62 for classifiers A (red dotted line) and B (blue line), respectively. *Classifier A* performs best around 50% of test cases executed, finding 79% of the failures. *Classifier B* has a very steep curve in the beginning, finding 57% of failures after executing less than 25% of the test cases. Differences in classifiers stem from different test cases used as input and are hard to explain as the training data is selected by the expert based on his expert knowledge. The resulting classifiers are hard to interpret and comprehend due to large feature spaces and a mathematical representation of the classifier. Consequently, when test execution is aborted due to time or resource constraints, our technique ensures that more of the failures are found, which is an improvement compared to the manual test execution order. Hence, our technique provides considerable improvement regarding effectiveness compared to random and manual execution order. Moreover, the results show that our technique is applicable to real-world data, providing better results than the currently applied manual approach. This indicates that our approach is able to find effective test cases important for the tested software version.

RQ2. Our technique allows to select arbitrary attributes (or their feature representation, respectively) as input for the learning process, i.e., we can combine different attributes with each other. Given the results of RQ1 and the fact that we include a variety of different classification attributes, it is of superior interest to identify those attributes that specifically contribute to the best prioritization results. As we performed different runs using all different attribute combinations, we are able to analyze each combination in detail. When reviewing the results for both subject systems in detail, we observed that the TCD had a significant impact on the prioritization of test cases. Hence, in the following we particularly distinguish between the average APFD values for combinations with and without using TCDs. The results for these combinations are shown in Fig. 2 and Fig. 3, respectively. We show the best and worst APFD values for the combinations in Table I. Moreover, we also discuss these results in relation to training and classification time, shown in Table II.

Body Comfort System. Our results show that using TCDs improves the failure detection rate significantly for BCS. In particular, our prioritization scores an average APFD value of 0.49 for attribute combinations without the TCD and an APFD

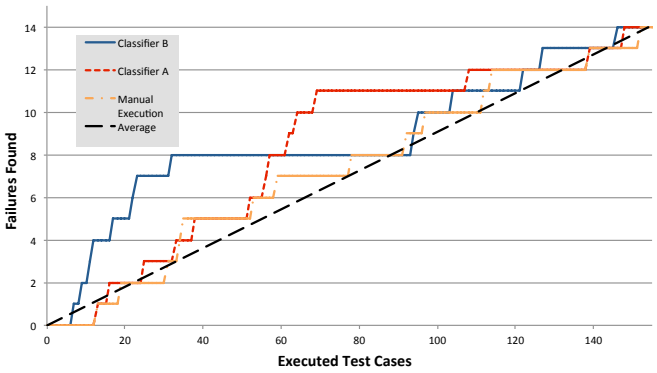

Fig. 4: Comparison of Test Expert vs. Machine Learning

TABLE I: Best and worst APFD results associated with corresponding attributes

APFD	BCS	Industry
Best	0.81 TCD	0.92 FC + EC + TCD + RC
Worst	0.38 FS,FA,FC	0.36 FA, FC, FP

TCD = Test Case Description, **FC** = Failure Count, **FA** = Failure Age, **FP** = Failure Priority, **EC** = Execution Cost, **RC** = Requirements Coverage

TABLE II: Overview of training and classification times

	BCS	Industry
Training		
Average of all Combinations with TCD	47.1 ms	2.9 s
Average of all Combinations without TCD	8.6 ms	127 ms
Classification		
Average of all Combinations with TCD	5.4 ms	134.7 ms
Average of all Combinations without TCD	0.6 ms	8.7 ms

of 0.74 with TCD. The best APFD value is achieved using TCD only for prioritization, while the worst result is achieved without it (cf. Fig. 2). Reasons for this are potential similarities between test cases according to their TCD, i.e., we are able to detect if test cases test the same parts of the systems. Thus, if one test case reveals a fault, another test cases covering the same part of the system might also reveal a failure. As a side effect, the training time with TCD is about 5 times higher than training time without TCD. Similarly, the classification time differs by a magnitude of nearly 10. Nevertheless, even in the worst case, time needed for training and classification, respectively, is in the range of milliseconds. Hence, we argue that the aforementioned differences are negligible.

Industry Data. For this data set, we are especially interested in the effects using the TCD because the natural language description is very common in practice and, thus, of great interest for the applicability of our technique. To this end, we show the average APFD values for combinations with and without TCDs as boxplots in Fig. 3.

Our data reveals that the prioritization achieves an average APFD value of 0.64 without and 0.85 with the TCD attribute,

thus, results are even better than for the BCS system. The peak value of 0.92 is achieved as a combination of TCD together with three further attributes (cf. Table I), while TCD is again not part of the worst result. Hence, the difference between the combinations with TCD attribute is even more significant than for all combinations without it (cf. Fig. 3). As explained for BCS, TCD allows to reveal test cases that test similar behavior, allowing to find more failures in the same subsystems.

Considering the training and classification times, we observed that a larger dictionary and number of requirements has a considerable impact. In particular, the prioritization needs at least 127 ms to train the classification model without using the TCD. If we take the TCD into account as input for learning the classifier, the training time increases up to 2.7 seconds. We measured these times using a notebook with two 2.9 GHz cores and 8 GB of RAM.

Overall, we constitute that the TCD has a positive impact on the prioritization in terms of APFD, and the resulting increased time for training and classification is negligible because it is in the range of seconds, while a manual prioritization of such quantities of test cases is infeasible. Nevertheless, durations must be evaluated for even larger or different data sets.

Threats to Validity. For RQ1, we use already seeded failures for BCS and already reported failures for industrial data as we did neither have access nor expertise to test the SUT. Hence, our approach can only be as good as the current failures indicate even though more unknown failures might have been revealed using our technique. We did not compare our technique to other existing prioritization techniques besides manual and random execution as these use different attributes for prioritization, e.g., historical data of specific runs, code-level information, or risk values. This threat is mitigated as we have shown that our technique is able to outperform a baseline (random) approach and the currently performed (manual) technique on real-world data, which shows the potential and applicability of our technique.

This comparison may exhibit threats regarding the test execution process as the test engineer groups test cases to be executed together, e.g., based on their preconditions. However, we argue that for our evaluation, the model has been trained by the expert and, thus, constitutes an order that is inspired by the tester's decisions and is comparable to the current, manual execution. Furthermore, as we use supervised ML, the selection of training data is very important and influences the quality of the results. Different test experts might classify training data differently and, thus, introduce a threat to validity. However, our evaluation indicates that test experts are eligible to properly train the ML to achieve good APFD values. Even though one industrial case study is not enough to generalize the results, it indicates the potentials and applicability of our technique on real data sets.

While we compared our technique to random prioritization in terms of computation time in RQ2, testers usually do not prioritize their test cases, but perform an ad-hoc selection which depends on the current context and project situation. Additionally, the manual prioritization of several hundred test cases is very time consuming and non-trivial, rendering a comparison almost infeasible. However, we argue that the speed of the computation is very fast and, thus, is applicable in a practical scenario.

V. Related Work

In this section, we acknowledge prior work on test case prioritization and machine learning-based testing techniques. The existing techniques differ to our approach in their used artifacts (i.e., either only subsets of our technique or other data, such as code or severity values) and the underlying concepts (i.e., most are not applying machine learning). In contrast to our novel approach, none of the existing techniques are able to analyze and exploit test case descriptions in an automated fashion. We present related work according to their used artifacts in the following.

Cost-based Prioritization and Selection: Wong et al. [32] were one of the first to propose the prioritization idea for testing. Their approach uses increasing cost per additional coverage of modified code and, thus, is applicable only for white-box testing. Herzig et al. [18] present a self-adaptive test case selection strategy called THEO, which is based on execution costs of test cases. Their approach automatically skips test executions if a test case's expected execution cost exceeds the cost for not executing them. The input information is available in black-box domains as THEO only requires historical test data, such as execution time, and context information, such as the build type and architecture. In contrast to us, they select or prioritize test cases based on different cost functions.

Requirement-Based Prioritization. Within the black-box domain, requirements-based testing is often performed to prioritize test cases. Srikanth et al. [28] present the *PORT* prioritization technique for system test cases based on four different factors: customer-assigned priority of requirements, developer-perceived implementation complexity, requirements volatility and fault proneness of requirements. Moreover, Chen and Wang [8] present a test case prioritization technique based on a requirements severity score and inter-case dependency. They use both a genetic algorithm and ant colony optimization to prioritize test cases. Perini et al. [23] present the case-based ranking (CBRank) method to prioritize requirements. It uses information by the project's stakeholder and priority values learned by a machine learning approach.

History-Based Prioritization. Qu et al. [25] present a test case prioritization approach for black-box systems based on test history and run-time information, which leads to the computation of a test case relation matrix. Engström et al. [12] introduce a history-based prioritization of test cases based on work by Fazlalizadeh [14]. Their approach uses different black-box prioritization factors, such as cost, execution history, static priority, and test case age. Similar to our work, Engström et al. evaluated their approach based on data in QC and measured the failure detection rate. While these techniques are applicable to black-box testing and a qualitative comparison to our technique should be performed, we could not perform a comparison due to two reasons: Most of the techniques require

different data, such as requirements severity, run-specific test case history or stakeholder information. In addition, some tools have not been available.

Machine Learning-Based Prioritization. Machine-learning based testing approaches are often based on source code information. These fault-prediction techniques have been surveyed by Catal [7] showing that ML approaches can improve the probability of fault detection compared to classic software reviews using code-specific metrics. Briand [5] presents an overview of applications for ML in software testing, e.g., in risk-driven testing or as test oracles. To realize this, Briand et al. [6] present the MELBA methodology that abstracts black-box test suite information using the Category-Partitioning [22] method. Moreover, they use Decision Trees to learn about the relationship between test input and output equivalence classes to find potential redundancies. In contrast to MELBA, we prioritize test cases and use natural language test cases instead of abstractions.

VI. Conclusion and Future Work

We presented a novel technique for test case prioritization in system-level testing without code access. It utilizes high-level artifacts (i.e., test cases, requirements and failures described in natural language) for black-box testing. We apply the supervised ML technique SVM Rank to learn a ranked classification model. We evaluated our technique on two subject systems to assess the capability of early failure detection capabilities using the APFD metric. Our results reveal considerable improvements compared to a random and manual prioritization by test experts. Our technique is able to find failures earlier and, thus, allowing for more efficient regression testing. Including natural language artifacts yields considerable improvements in terms of failure detection rate.

For future work, we want to assess further ML techniques for black-box test case prioritization. Among others, we suggest to use Artificial Neural Networks or Info-Fuzzy Networks for black-box testing as they have been successfully used as white-box test oracles in literature [2]. We plan to take further meta-data into account as attributes for the classification model (e.g., requirements priority or risk information). Also, further studies will allows for generalizing results. To guide the prioritization process, more investigations in the selection of training data for this particular domain have to be performed. Finally, a boosting approach [31] to reuse classification models from previous learning phases is a promising future direction to further improve the prioritization results.

References

[1] ISO/DIS 26262-1 - Road vehicles - Functional safety, 2009.

[2] D. Agarwal, D. Tamir, M. Last, and A. Kandel. A comparative study of artificial neural networks and info-fuzzy networks as automated oracles in software testing. 42(5):1183–1193, 2012.

[3] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[4] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., 2006.

[5] L. Briand. Novel applications of machine learning in software testing. In *Proc. Int'l Conf. Quality Software*, pages 3–10, 2008.

[6] L. Briand, Y. Labiche, and Z. Bawar. Using machine learning to refine black-box test specifications and test suites. In *Proc. Int'l Conf. Quality Software*, pages 135–144, 2008.

[7] C. Catal. Software fault prediction: A literature review and current trends. 38(4):4626–4636, 2011.

[8] G. Chen and P.-Q. Wang. Test case prioritization in a specification-based testing environment. 9(8), 2014.

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

[10] J. Czerwonka, R. Das, N. Nagappan, A. Tarvo, and A. Teterev. Crane: Failure prediction, change analysis and test prioritization in practice – experiences from windows. In *Proc. Int'l Conf. Software Testing, Verification, and Validation*, pages 357–366. IEEE, 2011.

[11] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. *Proc. Int'l Conf. Software Engineering*, 2001.

[12] E. Engström, P. Runeson, and A. Ljung. Improving regression testing transparency and efficiency with history-based prioritization – an industrial case study. In *Proc. Int'l Conf. Software Testing, Verification, and Validation*, pages 367–376. IEEE, 2011.

[13] E. Engström, P. Runeson, and M. Skoglund. A systematic review on regression test selection techniques. 52:14–30, 2010.

[14] Y. Fazlalizadeh, A. Khalilian, M. A. Azgomi, and S. Parsa. Prioritizing test cases for resource constraint environments using historical test case performance data. pages 190–195. IEEE, 2009.

[15] K. Gao, T. M. Khoshgoftaar, and A. Napolitano. A hybrid approach to coping with high dimensionality and class imbalance for software defect prediction. pages 281–288, 2012.

[16] M. J. Harrold. Testing: A roadmap. In *Proc. Int'l Conf. Software Engineering*, pages 61–72. ACM, 2000.

[17] M. J. Harrold, D. Liang, and S. Sinha. An approach to analyzing and testing component-based systems. In *ICSE'99 Workshop on Testing distributed Component-Based Systems*, 1999.

[18] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *Proc. Int'l Conf. Software Engineering*. IEEE, 2015. to appear.

[19] T. Joachims. Optimizing search engines using clickthrough data. pages 133–142. ACM, 2002.

[20] D. E. King. Dlib-ml: A machine learning toolkit. 10:1755–1758, 2009.

[21] S. Lity, R. Lachmann, M. Lochau, and I. Schaefer. Delta-oriented software product line test models - the body comfort system case study. Technical report, TU Braunschweig, 2013.

[22] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Comm. ACM*, 31(6):676–686, 1988.

[23] A. Perini, A. Susi, and P. Avesani. A machine learning approach to software requirements prioritization. *IEEE Trans. Soft. Eng.*, 39(4):445–461, 2013.

[24] M. F. Porter. Readings in information retrieval. pages 313–316. Morgan Kaufmann Publishers Inc., 1997.

[25] B. Qu, C. Nie, B. Xu, and X. Zhang. Test case prioritization for black box testing. In *Proc. Int'l Comp. Soft. and Appl. Conf.*, pages 465–474, 2007.

[26] G. Rothermel and M. Harrold. Analyzing regression test selection techniques. *IEEE Trans. Soft. Eng.*, 22(8):529–551, 1996.

[27] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Trans. Soft. Eng.*, Vol.27 No.10:929–948, 2001.

[28] H. Srikanth, S. Banerjee, L. Williams, and J. Osborne. Towards the prioritization of system test cases. 24(4):320–337, 2014.

[29] M. E. Tipping. Sparse bayesian learning and the relevance vector machine. 1:211–244, 2001.

[30] J. J. Webster and C. Kit. Tokenization as the initial phase in nlp. pages 1106–1110. Association for Computational Linguistics, 1992.

[31] I. H. Witten, E. Frank, and M. A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., 3rd edition, 2011.

[32] W. E. Wong, J. R. Horgan, S. London, and H. A. Bellcore. A study of effective regression testing in practice. pages 264–274. IEEE, 1997.

[33] Z. Yao and C. Ze-wen. Research on the construction and filter method of stop-word list in text preprocessing. volume 1, pages 217–221, 2011.

[34] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. 22(2):67–120, 2007.