Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Original software publication

Cost-effective simulation-based test selection in self-driving cars software

Christian Birchler[a,*], Nicolas Ganz[a], Sajad Khatiri[b,a], Alessio Gambi[c], Sebastiano Panichella[a]

[a] *Zurich University of Applied Sciences, Switzerland*
[b] *Software Institute - USI Lugano, Switzerland*
[c] *University of Passau, Germany*

A R T I C L E   I N F O

Article history:
Received 14 April 2022
Received in revised form 11 January 2023
Accepted 12 January 2023
Available online 18 January 2023

Keywords:
Self-driving cars
Software simulation
Regression testing
Test case selection
Continuous integration

A B S T R A C T

Simulation environments are essential for the continuous development of complex cyber-physical systems such as self-driving cars (SDCs). Previous results on simulation-based testing for SDCs have shown that many automatically generated tests do not strongly contribute to the identification of SDC faults, hence do not contribute towards increasing the quality of SDCs. Because running such “uninformative” tests generally leads to a waste of computational resources and a drastic increase in the testing cost of SDCs, testers should avoid them. However, identifying “uninformative” tests *before* running them remains an open challenge. Hence, this paper proposes SDC-Scissor, a framework that leverages Machine Learning (ML) to identify SDC tests that are unlikely to detect faults in the SDC software under test, thus enabling testers to skip their execution and drastically increase the cost-effectiveness of simulation-based testing of SDCs software. Our evaluation concerning the usage of six ML models on two large datasets characterized by 22’652 tests showed that SDC-Scissor achieved a classification F1-score up to 96%. Moreover, our results show that SDC-Scissor outperformed a randomized baseline in identifying more failing tests per time unit.

Webpage & Video: https://github.com/ChristianBirchler/sdc-scissor

* Corresponding author.
E-mail address: birchler.chr@gmail.com (C. Birchler).

https://doi.org/10.1016/j.scico.2023.102926

Metadata

Table 1
Code metadata.

Code metadata description	
Current code version	v2.1.2
Permanent link to code/repository used for this code version	https://github.com/ScienceofComputerProgramming/SCICO-D-22-00093
Legal Code License	GNU General Public License (GPLv3)
Code versioning system used	Git
Software code languages, tools, and services used	Python 3.9, BeamNG.tech v0.24.0.2
Compilation requirements, operating environments and dependencies	Windows 10
If available, link to developer documentation/manual	https://sdc-scissor.readthedocs.io/en/latest/
Support email for questions	birchler.chr@gmail.com, spanichella@gmail.com

1. Introduction

Cyber-physical systems (CPSs) are complex systems that leverage physical capabilities from hardware components [1] and find applications in various domains including Robotics, Transportation and Healthcare. For instance, in the automotive domain, self-driving cars (SDCs) are one emerging example of CPS, expected to impact the transport system of our society profoundly. Specifically, human driving errors cause more than 90% of car accidents [2] and SDCs have the potential to reduce such errors and eliminate most of these accidents. However, the recent fatal crashes involving SDCs suggest that the advertised large-scale adoption of SDCs appears optimistic [1].

Automated testing of SDCs (and in general CPS) to ensure their proper behavior is still an open research challenge [3]. We argue that enabling cost-effective testing automation in Continuous Integration (CI) pipelines for SDCs is critical to address the safety and reliability requirements of SDCs [2,4]. However, current SDC testing practices have several limitations: (i) difficulty in testing SDCs using representative, safety-critical tests [5]; (ii) difficulty in assessing SDC's behavior in different environmental conditions [2].

To deal with such safety-related challenges, there is an increasing interest in adopting agile development paradigms within the CPS safety-critical domains [6,7] to identify hazards and elicit safety requirements iteratively [8]. Consequently, researchers proposed the usage of Digital-Twins[1] technologies to simulate and test CPSs in a diversified set of scenarios [9–13] to support testing automation [14,15], regression testing [12,16], and debugging [17,18] activities. In this context, simulation-based testing has been suggested as a promising direction to improve the SDC testing practices [19–21] because simulation environments enable efficient test execution, reproducible results, and testing under critical conditions [22]. Additionally, simulation-based testing can be as effective as traditional field operational testing [3,23]. However, the testing space of simulation environments is infinite, which poses the challenge of exercising the SDC behaviors adequately [24,25]. Given the limited budget devoted to testing activities, it is paramount that developers test SDCs in a cost-effective fashion: using test suites optimized to reduce testing effort (time) without affecting their ability to identify faults [26,27,25].

To increase SDC testing cost-effectiveness, we propose **SDC-Scissor** (**SDC** co**S**t-effe**C**t**I**ve te**S**t **S**elect**OR**), a framework that leverages Machine Learning (ML) approaches for identifying tests that are unlikely to detect faults and skips them before their execution, hence, reducing the time spent in executing tests. Specifically, we refer to tests that do not expose a fault as *safe* and deem them irrelevant. On the contrary, we consider tests that expose a fault (e.g., an SDC drives out of the road) as relevant and refer to them as *unsafe*.

SDC-Scissor exploits six ML models trained on SDC simulation-based tests features that can be computed before the actual test execution (i.e., input features) to classify whether the tests are safe or unsafe [12,16].

We originally proposed employing Machine Learning to classify simulation-based tests and select them in [12] for making the testing of SDCs more cost-effective. This paper extends our original work by making the following contributions:

- A structural refactoring and extension of SDC-Scissor framework to provide an extendable open API (e.g., facilitating the integration of other SDC simulation environments, or an interface to implement an own AI or an own test generator) as well as the possibility of using the z coordinate (defining a road position in three-dimensional space), which increases the level of realism of generated tests (given the non-flat roads).
- An extension of original datasets that include new configurations of the test subject (i.e., risk factors RF1, RF1.5, and RF2) and additional 14'107 simulations-based tests.
- We extended the automated, ML-based approach integrating more ML models trained on features of SDC simulation-based tests to classify whether SDC tests are safe or unsafe (computed before the actual test execution);
- An empirical study comparing the cost-effectiveness of the proposed approach with a randomized baseline as well as a Mean Decrease in Gini analysis to describe the most important SDC features used by the ML models in identifying unsafe tests.

[1] A digital twin is a virtual representation of a real-time digital counterpart of a physical object or process.

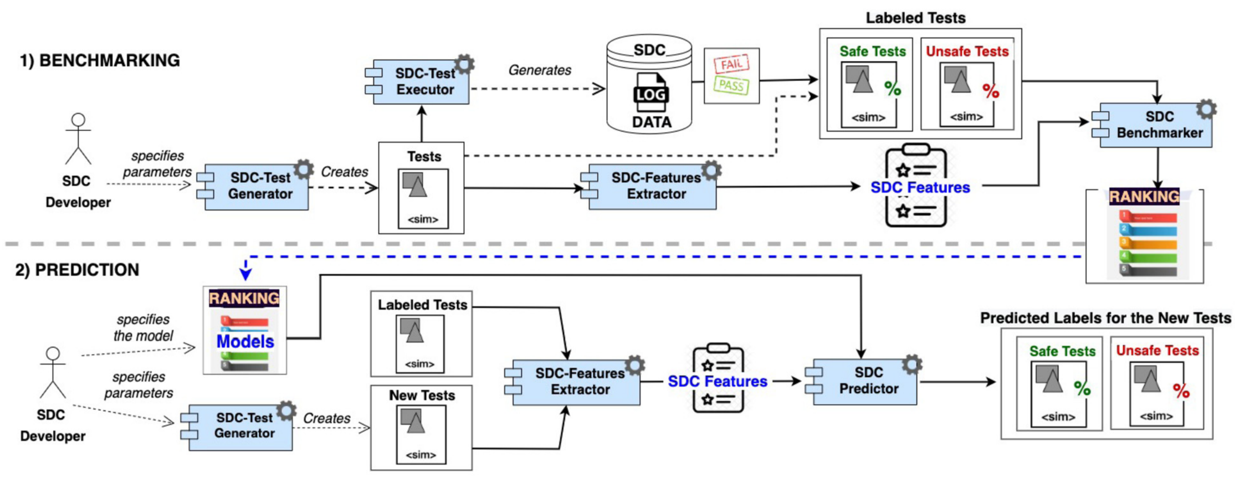

Fig. 1. The SDC-Scissor's architecture.

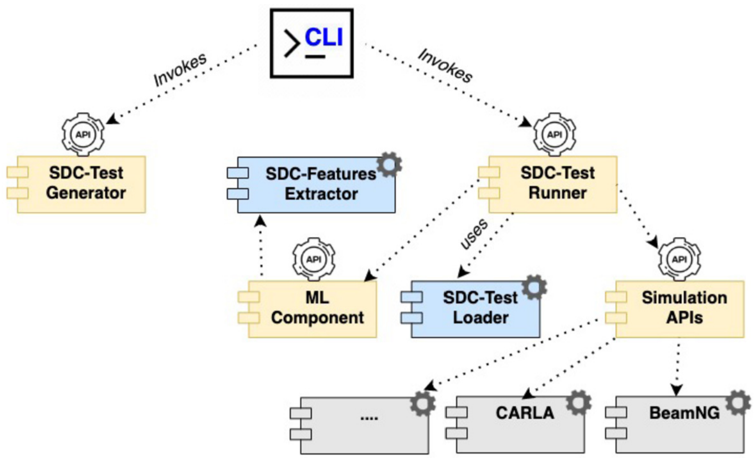

Fig. 2. The SDC-Scissor's main APIs.

- To enable future studies, we made SDC-Scissor compatible with the recent version of BeamNG.tech (v0.24.0.2), which allows the generation of more diverse tests, with the possibility to test multiple cars simultaneously.

Through a large empirical study concerning the usage of six ML models on two large datasets characterized by around 23'000 SDC simulation-based tests, we assessed the performance of SDC-Scissor in optimizing simulation-based testing. Our evaluation showed that SDC-Scissor achieved a higher classification F1-score (between 56% and 96%) with the best performing ML models and outperformed a randomized baseline in identifying failing tests as well as in reducing the time spent running uninformative (i.e., safe) tests.

2. The SDC-Scissor tool

In this section, we give an overview of SDC-Scissor's software architecture and its main usage scenarios (Fig. 1); we describe the simulation environment it uses (i.e., BeamNG.tech) and its main APIs (Fig. 2); finally, we discuss in details the components, the approach and the technologies behind SDC-Scissor.

2.1. SDC-Scissor architecture overview & main scenarios

SDC-Scissor supports two main usage scenarios: *Benchmarking* and *Prediction*. In the *Benchmarking* scenario, developers leverage SDC-Scissor to determine the best ML model(s) to classify SDC simulation-based tests as safe or unsafe. In the *Prediction* scenario, instead, developers use those model(s) to classify and select newly generated test cases.

SDC-Scissor Software Architecture implements these scenarios by means of the following software components (Fig. 1): (i) `SDC-Test Generator` generates random SDC simulation-based tests, and (ii) `SDC-Test Executor` executes them.

Table 2
Full Road Attributes extracted by the *SDC-Features Extractor*.

Feature	Description	Range
direct_distance	Euclidean dist. between start and end (m)	[0 – 490]
road_distance	Tot. length of the driving path (m)	[50.6 – 3,317]
num_l_turns	Nr. of left turns on the driving path	[0 – 18]
num_r_turns	Nr. of right turns on the driving path	[0 – 17]
num_straights	Nr. of straight segments on the driving path	[0 – 11]
total_angle	Cumulative turn angle on the driving path	[105 – 6,420]

The test results produced by `SDC-Test Executor` are recorded and used to label tests as safe or unsafe; (iii) `SDC-Features Extractor` extracts input features of the executed SDC tests, while (iv) `SDC-Benchmarker` uses these features and corresponding labels as input to train the ML models and determine which model best predicts the tests that are more likely to detect faults in SDCs; finally, (v) `SDC-Predictor` uses the ML models to classify newly generated test cases and enables test selection.

2.2. BeamNG.tech's simulation environment

SDC-Scissor uses BeamNG.tech to execute SDC tests as physically accurate and photo-realistic driving simulations. BeamNG.tech can procedurally generate tests [24] and was recently adopted in the ninth edition of the Search-Based Software Testing (SBST) CPS testing tool competition [28].

BeamNG.tech is organized around a central *game engine* that communicates with the *physics simulation*, the *UI*, and the *BeamNGpy API*.[2] The UI can be used for game control and manual content creation (e.g., *assets*, *scenarios*). For example, developers can use the world editor to create or modify the virtual environments that are used in the simulations; testers, instead, can create test scripts implementing driving scenarios (i.e., the tests). The API, instead, allows the automated generation and execution of tests, the collection of simulation data (e.g., camera images, LIDAR point clouds) for training, testing, and validating SDCs. It also enables driving agents to drive simulated vehicles and get programmatic control over running simulations (e.g., pause/resume simulations, move objects around). The *game engine* manages the simulation setup, camera, graphics, sounds, gameplay, and overall resource management. The *physics core*, instead, handles resource-intensive tasks such as collision detection and basic physics simulation; it also orchestrates the concurrent execution of the vehicle simulators. The *vehicle simulators* —one for each of the simulated vehicles— simulate the high-level driving functions and the vehicle sub-systems (e.g., drivetrain, ABS).

We employ the BeamNG.AI[3] lane-keeping system as the test subject for our evaluation: the driving agent is shipped with BeamNG.tech and drives the car by computing an ideal driving trajectory to stay in the center of the lane while driving within a configurable speed limit. As explained by BeamNG.tech developers, the *risk factor* (RF) is a parameter that controls the driving style of BeamNG.AI: low-risk values (e.g., 0.7) result in smooth driving, whereas high-risk values (e.g., 1.7 and above) result in an edgy driving that may lead the ego-car to cut corners [12].

2.3. The SDC-Scissor's approach and technology overview

SDC-Scissor integrates the extensible testing pipeline defined by the SBST tool competition[4] in its `SDC-Test Executor`. We use the SBST tool competition infrastructure since it allows us to (i) seamlessly execute the tests in BeamNG.tech and (ii) distinguish between *safe* and *unsafe* tests based on whether the self-driving car keeps its lane (non-faulty tests) or depart from it (faulty tests) [24]. Consequently, SDC-Scissor can accommodate various `SDC-Test Generators` for generating SDC simulation-based tests. In this paper, we demonstrate SDC-Scissor by using the Frenetic test generation [29], one of the most effective tool submitted to the SBST tool competition.

SDC-Scissor predicts whether the tests are likely to be safe or unsafe before their execution using input features that `SDC-Features Extractor` extracted. Specifically, this component extracts *Full Road Features* (FRFs), i.e., a set of SDC features that describe the global characteristics of the tests. Those features include the main *road attributes* (see Table 2) and *road statistics* concerning the road composition (see Table 3). Road statistics are calculated in three steps: (i) extraction of the *reference driving path* that the ego-car has to follow during the test execution (e.g., the road segments that the car needs to traverse to reach the target position); (ii) extraction of metrics available for each road segment (e.g., length of road segments); and (iii) computation of standard aggregation functions on the collected road segments metrics (e.g., minimum and maximum).

SDC-Scissor relies on the `SDC-Benchmarker` to determine the ML model that best classifies the SDC tests that are likely to detect faults. It follows an empirical approach to do so: given a set of labeled tests and corresponding input fea-

[2] `beamngpy` is available on PyPI and Github (https://github.com/BeamNG/BeamNGpy).
[3] https://wiki.beamng.com/Enabling_AI_Controlled_Vehicles#AI_Modes.
[4] https://github.com/se2p/tool-competition-av.

Table 3
Full Road Statistics extracted by the *SDC-Features Extractor*.

Feature	Description	Range
median_angle	Median turn angle on the driving path (DP)	[30 – 330]
std_angle	Std. Deviation of turn angles on the DP	[0 – 150]
max_angle	Max. turn angle on the DP	[60 – 345]
min_angle	Min. turn angle on the DP	[15 – 285]
mean_angle	Average turn angle on the DP	[52.5 – 307.5]
median_pivot_off	Median turn radius on the DP	[7 – 47]
std_pivot_off	Std. Deviation of turn radius on the DP	[0 – 22.5]
max_pivot_off	Max. turn radius on the DP	[7 – 47]
min_pivot_off	Min. turn radius on the DP	[2 – 47]
mean_pivot_off	Average turn radius on the DP	[5.3 – 47]

tures, `SDC-Benchmarker` trains and evaluates an ensemble of standard ML models using the well-established `sklearn`[5] library. Next, it assesses ML models' quality using either 10-fold cross-validation or a testing dataset; and, finally, selects the best performing ML models according to Precision, Recall, and F1-score metrics [12]. Noticeably, SDC-Scissor can use many different ML models; however, in this work, we consider Naive Bayes [30], Logistic Regression [31], Random Forests [32], Gradient Boosting [33], Support Vector Machine [34], and Decision Tree [35]. We do so because these ML models have been successfully used for defect prediction or other classification problems in Software Engineering [36,37].

Finally, the `SDC-Predictor` uses the ML models to predict the likelihood that newly generated SDC tests are safe or not. Specifically, developers have the possibility to select the ML models recommended by the `SDC-Benchmarker` (considered most accurate), or they can select other models of their choice.

2.4. SDC-Scissor's main APIs

SDC-Scissor was refactored and is now more modularized into components that offer APIs for enhancing better extensibility of the tool, as shown in Fig. 2. The CLI component is where the user directly interacts with the tool, as described in Section 3. Furthermore, other test generators can be integrated by implementing the relevant API of the `SDC-Test Generator` component. The main goal of the refactoring was to enable SDC-Scissor to work with other simulators for the future (e.g., CARLA). For this purpose, we define `Simulation APIs` for simulators. The current version of SDC-Scissor provides an implementation of the API for the BeamNG.tech simulator. SDC-Scissor also provides a `ML Component` and API for the training and testing of the machine-learning models. This allows SDC-Scissor to experiment easier on more diverse test selection approaches for the research on simulation-based regression testing on SDCs.

3. Using SDC-Scissor

SDC-Scissor tool is openly available and can be used as a Python command-line utility via `poetry`[6] or `pip`. In the following sections, it will be explained how SDC-Scissor can be installed and used for *Benchmarking* and *Prediction* as shown in Fig. 1.

3.1. Installation

```
git clone https://github.com/ChristianBirchler/sdc-scissor.git
cd sdc-scissor
poetry install
poetry run sdc-scissor [COMMAND] [OPTIONS]
```

To simplify SDC-Scissor's usage, we also enable to execute it as a Docker[7] container:

```
docker build --tag sdc-scissor .
docker run --volume "$(pwd)/results:/out" --rm
    sdc-scissor [COMMAND] [OPTIONS]
```

As we detail below, SDC-Scissor's command-line supports the execution of the main usage scenarios described in Section 2.2 by taking appropriate commands and inputs (see Fig. 3).

3.2. Benchmarking

Test generation. To generate SDC tests by running the Frenetic generator within a given number of desired tests, SDC-Scissor requires the following command:

[5] https://scikit-learn.org/.
[6] https://python-poetry.org/.
[7] https://www.docker.com.

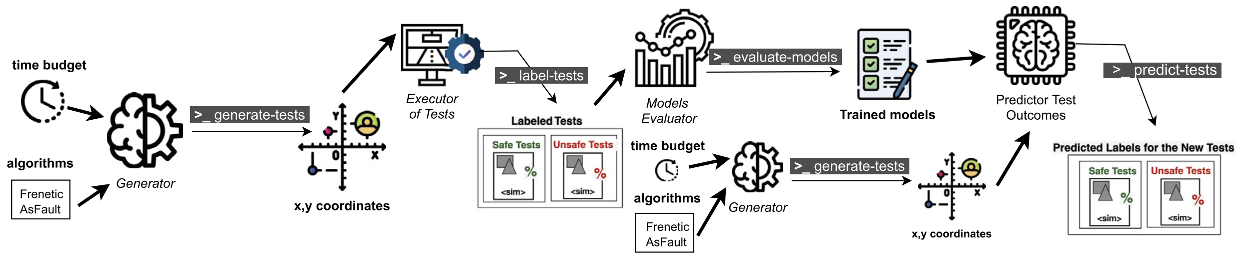

Fig. 3. The SDC-Scissor's fine-grained view.

```
poetry run sdc-scissor generate-tests -c {number of tests to generate}
```

Automated test labeling. SDC-Scissor labels tests as safe and unsafe by executing them in BeamNG.tech. Since BeamNG.tech cannot be run as a Docker container, labeling tests can only be run locally (i.e., outside Docker). This labeling facility allows developers to create datasets that can be used for the training and validation of ML models (e.g., ML-based prediction of unsafe tests) in the context of *Benchmarking*. Generating a labeled dataset requires a set of already generated SDC tests and the execution of the following command:

```
poetry run sdc-scissor label-tests -t /path/to/tests --rf {risk factor} --oob {OOB criteria}
```

If the car drives out of the lane to a certain percentage, also referred to as the out-of-bound (OOB) criteria, then the test is labeled as unsafe. Based on the arguments for the risk factor and OOB, the tests will be labeled. With different values for those arguments, the tests can be labeled differently and, therefore, also affect the ML-based predictions.

Feature extraction. The ML models require as inputs features as described in Table 2 and Table 3. SDC-Scissor extracts those features from the tests and stores them in a separate CSV file with the following command:

```
poetry run sdc-scissor extract-features -t /path/to/tests
```

ML models evaluation. For identifying the models that SDC-Scissor could use for the prediction, SDC-Scissor implements a 10-fold cross-validation strategy on the labeled dataset. The following command tells SDC-Scissor to benchmark all the configured ML models:

```
poetry run sdc-scissor evaluate-models --csv /path/to/road_features.csv
```

3.3. Prediction

For the prediction use case scenario, we generate new tests with the same command as in Section 3.2. The goal is to predict the test outcome before executing them. For this reason, we generate new tests for which we do not know the oracle yet.

Test outcome prediction. SDC-Scissor classifies unlabeled tests, i.e., it predicts their outcome, using a trained ML model with the following command:

```
poetry run sdc-scissor predict-tests -t /path/to/tests
```

Random baseline evaluation. SDC-Scissor allows to select tests using a random strategy that provides a baseline evaluation with the following command:

```
poetry run sdc-scissor evaluate-cost-effectiveness -csv /path/to/road_features.csv
```

4. Empirical evaluation

In this paper, we seek to answer the following research questions:

- RQ_1: *To what extent is it possible to predict safe and unsafe SDC test cases?*
- RQ_2: *To what extent SDC-Scissor is cost-effective compared to a random baseline?*
- RQ_3: *To what extent are different road features relevant to predict safe and unsafe SDC test cases?*

We are interested to investigate the extent to which predicting unsafe SDC test cases before executing them (RQ_1) is possible in a practical sense (e.g., do we achieve a reasonable precision, recall, and F-measure?). More importantly, we also investigate whether SDC-Scissor allows reducing testing cost dedicated to the execution of so-called irrelevant tests (RQ_2), i.e., test cases not leading to actual faults. To achieve these objectives, as described below, we, first of all, constructed a dataset of SDC test cases that can be used to experiment with such research questions. Hence, we specifically investigated the usage of SDC road features to predict SDC test outcomes as well as investigate the ability of SDC-Scissor to outperform a

Table 4
Datasets Summary.

Dataset	Test	Data Points		
	Subject	Unsafe	Safe	Total
Dataset 1	BeamNG.AI cautious	1’318 (28%)	3’397 (72%)	4’715
	BeamNG.AI moderate	1’502 (34%)	2’908 (66%)	4’410
	BeamNG.AI reckless	1’680 (34%)	3’302 (66%)	4’982
Dataset 2	BeamNG.AI cautious	312 (26%)	866 (74%)	1’178
	BeamNG.AI moderate	2’543 (45%)	3’095 (55%)	5’638
	BeamNG.AI reckless	1’655 (96%)	74 (4%)	1’729
	Total	9’010 (40%)	13’642 (60%)	22’652

Table 5
Performance of the best three ML models with dataset split 80/20. The best results are shown in boldface.

Dataset	RF	Model	Prec.	Recall	F1-score
Dataset 1	**RF 1**	Logistic Regression	**40.3%**	55.5%	**46.7%**
		Naïve Bayes	**40.3%**	49.8%	44.6%
		Random Forest	38.9%	**57.5%**	**46.4%**
Dataset 1	**RF 1.5**	Logistic Regression	**45.8%**	60.9%	52.3%
		Naïve Bayes	40.2%	**92.5%**	**56.1%**
		Random Forest	41.3%	30.5%	35.1%
Dataset 1	**RF 2**	Logistic Regression	**39.4%**	53.6%	45.5%
		Naïve Bayes	34.6%	**100.0%**	**51.4%**
		Random Forest	38.3%	53.3%	44.6%
Dataset 2	**RF 1**	Logistic Regression	**43.3%**	87.3%	**57.9%**
		Naïve Bayes	36.7%	**92.1%**	52.5%
		Random Forest	40.7%	79.4%	53.8%
Dataset 2	**RF 1.5**	Logistic Regression	78.1%	**65.3%**	**71.1%**
		Naïve Bayes	**79.3%**	53.2%	63.6%
		Random Forest	75.8%	62.7%	68.6%
Dataset 2	**RF 2**	Logistic Regression	**99.6%**	82.8%	90.4%
		Naïve Bayes	98.7%	**94.3%**	**96.4%**
		Random Forest	**99.7%**	92.7%	**96.1%**

random baseline. Finally, we also discuss the most important features (RQ_3) used for enabling the prediction in the context of our work.

Dataset construction. We evaluated SDC-Scissor conducting a large study on two datasets, referred to as *Dataset 1* and *Dataset 2*, that contain over 22,000 SDC tests (see Table 4). We adopted the following experimental setup to obtain comprehensive and unbiased training datasets. For *Dataset 1*, we *randomly* generated 13,207 valid tests using Frenetic [29] as well as collected input features and executed them to collect labels. For the *Dataset 2*, instead, we generated 8,545 tests using AsFault [24].

AI engine and risk factor considered. It is important to note that in executing all those tests, we experimented with different BeamNG.AI risk factors as it influences the ego-car driving style. Specifically, we considered three configurations: cautious (RF 1.0), moderate (RF 1.5), and reckless (RF 2.0) driver. Using different values for the risk factor enabled us to study the effectiveness of SDC-Scissor on various SDCs’ driving styles. We empirically validated our expectations by running the cautious, moderate, and reckless drivers to generate both *Dataset 1* and *Dataset 2* tests. From Table 4 we can observe that the number of unsafe tests increased with increasing values of BeamNG.AI’s risk factor. This result seems to suggest that the risk factor may influence the safety of BeamNG.AI and the outcome of tests. However, we would like to make the note that for *Dataset 1*, the ratio of safe (66%) and unsafe (34%) tests between moderate (RF 1.5) and reckless (RF 2.0) drivers is identical.

ML models and training process considered. To assess the performance of SDC-Scissor in optimizing simulation-based SDCs testing via test selection (i.e., in selecting unsafe tests before executing them), for both *Dataset 1* and *Dataset 2*, we experimented with the ML models mentioned in Section 2.3 trained and validated using an 80/20 data split.

4.1. Results

Prediction (RQ$_1$). As reported in Table 5, on *Dataset 1* SDC-Scissor accurately identified unsafe test cases, with F1-score ranging between 35.1% and 56.1%. On *Dataset 2*, instead, SDC-Scissor identified unsafe test cases with F1-score ranging between 52.5% and 96.4%.

Cost-effectiveness (RQ$_2$). In the context of regression testing, we want to select only relevant test scenarios so that the testing cost (execution time) is reduced. We evaluated the cost-effectiveness of SDC-Scissor by computing the ratio of selected unsafe test scenarios and the overall test execution time. Thus, the cost-effectiveness score is computed as

Table 6
Cost-effectiveness ($= \frac{\text{number of unsafe tests selected}}{\text{simulation time of all selected tests}}$) of SDC-Scissor against a random baseline on *Dataset 1* with *RF 1.5*.

Model	Cost-effectiveness	
	SDC-Scissor	Random Baseline
Random Forest	0.29%	0.22%
Gradient Boosting	0.40%	0.19%
SVM	0.35%	0.19%
Naive Bayes	0.24%	0.22%
Logistic Regression	0.37%	0.22%
Decision Tree	0.23%	0.26%

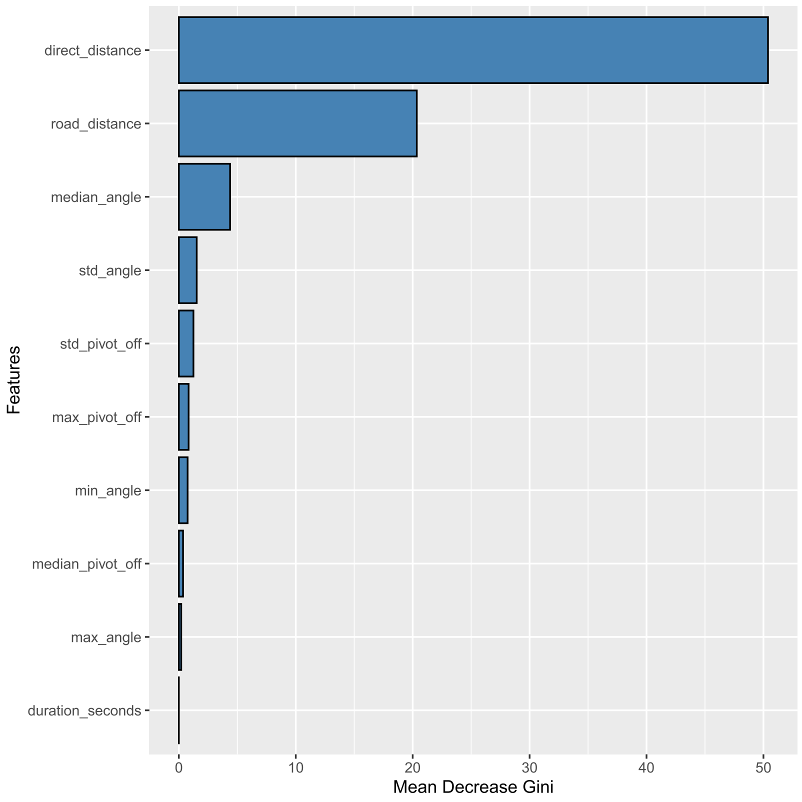

Fig. 4. Mean Decrease in Gini when using RF 1.0. The top 10 features are visualized (simulation time attributes included).

$$CE = \frac{\text{number of selected unsafe tests}}{\text{simulation time of all selected tests}}.$$

We compared the cost-effectiveness of SDC-Scissor with a random baseline test selector. In the case of SDC-Scissor, the models were trained on 80% of *Dataset 1 RF 1.5*. SDC-Scissor selected from the remaining 20% 10 tests that are most likely to be unsafe, whereas the random baseline selector picks 10 tests at random. As shown in Table 6, SDC-Scissor has only in the case of the *Decision Tree* model a worse cost-effectiveness of 0.23% against the baseline with a cost-effectiveness score of 0.26%. For the *Gradient Boosting*, *Support Vector Machine*, and *Logistic Regression* classifiers we have the highest differences of more than 0.1%. Overall, we observed a better cost-effectiveness score of SDC-Scissor compared to a random baseline test selector. In general, with our approach, we detect more unsafe tests as the baseline per time unit.

Feature Importance (RQ$_3$). To better understand the features that contribute more to the prediction of safe and unsafe tests, we computed the Mean Decrease in Gini (also called Mean Decrease in Impurity) [38–40] considering the designed road features. As shown in Fig. 4, Fig. 5 and Fig. 6, we can find the (top 10) features considered as important for the identification of safe and unsafe tests, for different risk factors (RF1.0, RF1.5, and RF2). It is interesting to observe from such features that the three top most important features vary depending on the specific configuration of the driving agent (i.e., RF). This observation suggests that certain characteristics of the road play an important role in the safety of the SDC, depending on the driving style (i.e., each RF). Specifically, for RF 1.0, the top three most important road features are the *Direct Distance*, *Road Distance*, and *Median Angle*. For RF 1.5, the top three most important road features are the *Road Distance*,

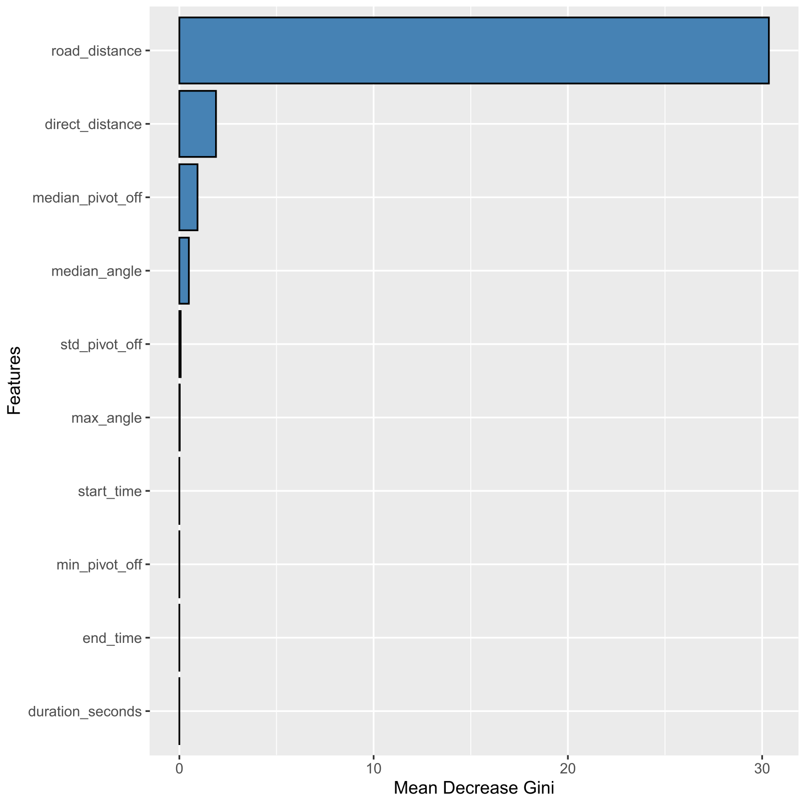

Fig. 5. Mean Decrease in Gini when using RF 1.5. The top 10 features are visualized (simulation time attributes included).

Direct Distance, and *Median Pivot Off*, while for RF 2.0, the top three most important road features are the *Road Distance*, *Direct Distance*, and *Mean Pivot Off*. Hence, for less cautious driving styles (for RF > 1.0), the most important feature is always represented by the *Road Distance*, followed by the *Direct Distance* feature and the *Mean/Median Pivot Off* feature. Finally, for a more cautious driving style (for RF $= 1.0$), the most important feature is represented by the *Direct Distance*, followed by the *Road Distance* and the *Median Angle* features. In a practical sense, this means that for a more cautious driving style (for RF $= 1.0$), the safety of the SDC is influenced by the direct/road distance and the turn angle on the driving path (i.e., the distance and the presence of turns are together influencing the SDC behavior). Complementary, for a less cautious driving style (for RF > 1.0), the safety of the SDC is influenced by the direct/road distance and the average/median radius of the road segments turned on the test track (i.e., the distance and the radius of specific road segments are together influencing the SDC behavior).

4.2. Threats to validity

SDC-Scissor is an ML-based test selector that depends on the data for training the models. The datasets were labeled with the internal BeamNG.AI of the used BeamNG simulator. The use of a single AI engine may introduce a threat to validity because the results might be biased since no other experiments with different AI were considered. Furthermore, we do not know how BeamNG.AI behaves with different weather conditions, which would increase the level of realism. The use of different simulators with different physical behavior could alter the results because BeamNG is a soft-body physics simulator with high fidelity that simulates deformations of multiple parts of the car, such as the chassis, engine, transmission, tires, etc., whereas other simulators like CARLA use a rigid-body physics engine. Furthermore, the ML models are trained with the default configurations. The prediction performances might be improved so that the results change and the ranking of the models vary.

5. Conclusions

This paper presented SDC-Scissor, an ML-based test selection approach that classifies SDC simulation-based tests as likely (or unlikely) to expose faults before executing them. SDC-Scissor trains ML models using input features extracted from driving scenarios, i.e., SDC tests, and uses them to classify SDC tests before their execution. Consequently, it selects only those tests that are predicted to likely expose faults. Our evaluation shows that SDC-Scissor successfully selected unsafe test

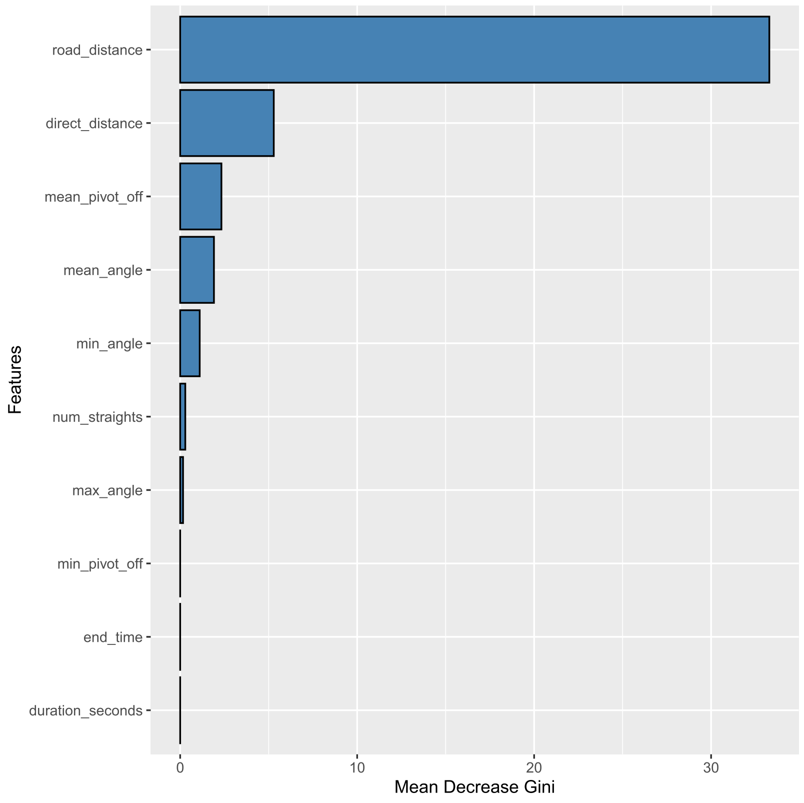

Fig. 6. Mean Decrease in Gini when using RF 2.0. The top 10 features are visualized (simulation time attributes included).

cases across different driving styles and drastically reduced the execution time dedicated to executing safe tests compared to a random baseline approach.

As future work, we plan to replicate our study on further SDC datasets, AI engines, and more advanced SDC features to study how the results generalize in various autonomous systems domains. Additionally, given our close contacts with the BeamNG.tech team, we plan the integration of SDC-Scissor into BeamNG.tech environment to enable researchers and SDC developers to use SDC-Scissor as a cost-effective testing environment for SDCs. Finally, we plan to investigate the use of SDC-Scissor in other CPS domains, such as drones, to investigate how it performs when testing focuses on different types of safety-critical faults. Specifically, it is important to investigate approaches that are more human-oriented or are able to integrate humans into-the-loop [36,37], via multi-objective optimizations [41,42].

Last but not least, our empirical research has some practical implications. It is our understanding that SDC-Scissor could be used in an industrial context to identify relevant test scenarios. When it comes to different levels of testing like Software-in-the-loop or Hardware-in-the-loop, SDC-Scissor provides a platform to conduct those experiments without manual human-based interaction. The testing costs can be reduced, and the fault detection rate is increased compared to a random test selector.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Christian Birchler reports financial support was provided by European Commission. Nicolas Ganz reports financial support was provided by European Commission. Sajad Khatiri reports financial support was provided by European Commission. Sebastiano Panichella reports financial support was provided by European Commission. Alessio Gambi reports financial support was provided by German Research Foundation.

Acknowledgements

We gratefully acknowledge the Horizon 2020 (EU Commission) support for the project *COSMOS* (DevOps for Complex Cyber-physical Systems), Project No. 957254-COSMOS) and the DFG project STUNT (DFG Grant Agreement n. FR 2955/4-1).

References

[1] R. Baheti, H. Gill, Cyber-physical systems, Impact Control Technol. 12 (1) (2011) 161–166.
[2] N. Kalra, S. Paddock, Driving to safety: how many miles of driving would it take to demonstrate autonomous vehicle reliability? Transp. Res., Part A, Policy Pract. 94 (2016) 182–193, https://doi.org/10.1016/j.tra.2016.09.010.
[3] A. Afzal, C. Le Goues, M. Hilton, C.S. Timperley, A study on challenges of testing robotic systems, in: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), IEEE, 2020, pp. 96–107.
[4] J. Kim, S. Chon, J. Park, Suggestion of testing method for industrial level cyber-physical system in complex environment, in: International Conference on Software Testing, Verification and Validation Workshops, 2019.
[5] F. Ingrand, Recent trends in formal validation and verification of autonomous robots software, in: International Conference on Robotic Computing, 2019, pp. 321–328.
[6] F. Zampetti, R. Kapur, M.D. Penta, S. Panichella, An empirical characterization of software bugs in open-source cyber-physical systems, J. Syst. Softw. 192 (2022) 111425, https://doi.org/10.1016/j.jss.2022.111425.
[7] A.D. Sorbo, F. Zampetti, C.A. Visaggio, M.D. Penta, S. Panichella, Automated identification and qualitative characterization of safety concerns reported in uav software platforms, ACM Trans. Softw. Eng. Methodol. (2022).
[8] J. Cleland-Huang, M. Vierhauser, Discovering, analyzing, and managing safety stories in agile projects, in: 26th IEEE International Requirements Engineering Conference, RE 2018, Banff, AB, Canada, August 20-24, 2018, 2018, pp. 262–273.
[9] Z. Huang, Y. Shen, J. Li, M. Fey, C. Brecher, A survey on ai-driven digital twins in industry 4.0: smart manufacturing and advanced robotics, Sensors 21 (19) (2021) 6340, https://doi.org/10.3390/s21196340.
[10] K. Bojarczuk, N. Gucevska, S.M.M. Lucas, I. Dvortsova, M. Harman, E. Meijer, S. Sapora, J. George, M. Lomeli, R. Rojas, Measurement challenges for cyber cyber digital twins: experiences from the deployment of Facebook's WW simulation system, in: F. Lanubile, M. Kalinowski, M.T. Baldassarre (Eds.), ESEM '21: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 11-15, 2021, ACM, 2021, pp. 2:1–2:10.
[11] A. Piazzoni, J. Cherian, M. Azhar, J.Y. Yap, J.L.W. Shung, R. Vijay, Vista: a framework for virtual scenario-based testing of autonomous vehicles, in: 2021 IEEE International Conference on Artificial Intelligence Testing, AITest 2021, Oxford, United Kingdom, August 23-26, 2021, IEEE, 2021, pp. 143–150.
[12] C. Birchler, N. Ganz, S. Khatiri, A. Gambi, S. Panichella, Cost-effective simulation-based test selection in self-driving cars software with sdc-scissor, in: The 29th IEEE International Conference on Software Analysis, Evolution, and Reengineering, 2022.
[13] V. Nguyen, S. Huber, A. Gambi, SALVO: automated generation of diversified tests for self-driving cars from existing maps, in: 2021 IEEE International Conference on Artificial Intelligence Testing, AITest 2021, Oxford, United Kingdom, August 23-26, 2021, IEEE, 2021, pp. 128–135.
[14] M. Alcon, H. Tabani, J. Abella, F.J. Cazorla, Enabling unit testing of already-integrated AI software systems: the case of apollo for autonomous driving, in: F. Leporati, S. Vitabile, A. Skavhaug (Eds.), 24th Euromicro Conference on Digital System Design, DSD 2021, Palermo, Spain, September 1-3, 2021, IEEE, 2021, pp. 426–433.
[15] F. Wotawa, On the use of available testing methods for verification & validation of ai-based software and systems, in: H. Espinoza, J. McDermid, X. Huang, M. Castillo-Effen, X.C. Chen, J. Hernández-Orallo, S.Ó. hÉigeartaigh, R. Mallah (Eds.), Proceedings of the Workshop on Artificial Intelligence Safety 2021 (SafeAI 2021) Co-Located with the Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2021), Virtual, February 8, 2021, in: CEUR Workshop Proceedings, vol. 2808, 2021, CEUR-WS.org, http://ceur-ws.org/Vol-2808/Paper_29.pdf, 2021.
[16] C. Birchler, S. Khatiri, P. Derakhshanfar, S. Panichella, A. Panichella, Single and multi-objective test cases prioritization for self-driving cars in virtual environments, ACM Trans. Softw. Eng. Methodol. (2022).
[17] S.C. Smith, S. Ramamoorthy, Attainment regions in feature-parameter space for high-level debugging in autonomous robots, in: IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2021, Prague, Czech Republic, September 27 - Oct, 1, 2021, IEEE, 2021, pp. 6546–6551.
[18] D. Roy, C. Hobbs, J.H. Anderson, M. Caccamo, S. Chakraborty, Timing debugging for cyber-physical systems, in: Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021, IEEE, 2021, pp. 1893–1898.
[19] A. Afzal, D.S. Katz, C. Le Goues, C.S. Timperley, Simulation for robotics test automation: developer perspectives, in: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST), IEEE, 2021, pp. 263–274.
[20] C.S. Timperley, A. Afzal, D.S. Katz, J.M. Hernandez, C.L. Goues, Crashing simulated planes is cheap: can simulation detect robotics bugs early?, in: 11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018, IEEE Computer Society, 2018, pp. 331–342, http://doi.ieeecomputersociety.org/10.1109/ICST.2018.00040.
[21] D. Wang, S. Li, G. Xiao, Y. Liu, Y. Sui, An exploratory study of autopilot software bugs in unmanned aerial vehicles, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 20–31.
[22] A. Gambi, T. Huynh, G. Fraser, Generating effective test cases for self-driving cars from police reports, in: Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM Press, 2019.
[23] A. Dosovitskiy, G. Ros, F. Codevilla, A.M. López, V. Koltun, CARLA: an Open Urban Driving Simulator, Conference on Robot Learning, vol. 78, Machine Learning Research, 2017, pp. 1–16, http://proceedings.mlr.press/v78/dosovitskiy17a.html.
[24] A. Gambi, M. Mueller, G. Fraser, AsFault: testing self-driving car software using search-based procedural content generation, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), IEEE, 2019.
[25] R.B. Abdessalem, S. Nejati, L.C. Briand, T. Stifter, Testing vision-based control systems using learnable evolutionary algorithms, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, 2018, pp. 1016–1026.
[26] S. Yoo, M. Harman, Using hybrid algorithm for Pareto efficient multi-objective test suite minimisation, J. Syst. Softw. 83 (4) (2010) 689–701.
[27] D.D. Nucci, A. Panichella, A. Zaidman, A.D. Lucia, A test case prioritization genetic algorithm guided by the hypervolume indicator, IEEE Trans. Softw. Eng. 46 (6) (2020) 674–696, https://doi.org/10.1109/TSE.2018.2868082.
[28] S. Panichella, A. Gambi, F. Zampetti, V. Riccio, Sbst Tool Competition 2021, International Conference on Software Engineering, Workshops, ACM, 2021.
[29] E. Castellano, A. Cetinkaya, C.H. Thanh, S. Klikovits, X. Zhang, P. Arcaini, Frenetic at the SBST 2021 tool competition, in: International Workshop on Search-Based Software Testing, IEEE, 2021, pp. 36–37.
[30] R. Caruana, A. Niculescu-mizil, An empirical comparison of supervised learning algorithms, in: Proc. 23 Rd Intl. Conf. Machine Learning (ICML'06, 2006, pp. 161–168.
[31] C. Sammut, G.I. Webb (Eds.), Logistic Regression, Springer US, Boston, MA, 2010, p. 631.
[32] T.K. Ho, The random subspace method for constructing decision forests, IEEE Trans. Pattern Anal. Mach. Intell. 20 (8) (1998) 832–844, https://doi.org/10.1109/34.709601.
[33] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, T.-Y. Liu, Lightgbm: a highly efficient gradient boosting decision tree, Adv. Neural Inf. Process. Syst. 30 (2017).
[34] S. Suthaharan, Support vector machine, in: Machine Learning Models and Algorithms for Big Data Classification, Springer, 2016, pp. 207–235.
[35] S.R. Safavian, D. Landgrebe, A survey of decision tree classifier methodology, IEEE Trans. Syst. Man Cybern. 21 (3) (1991) 660–674.
[36] S. Panichella, A.D. Sorbo, E. Guzman, C.A. Visaggio, G. Canfora, H.C. Gall, How can I improve my app? Classifying user reviews for software maintenance and evolution, in: International Conference on Software Maintenance and Evolution, IEEE, 2015, pp. 281–290.

[37] A. Di Sorbo, S. Panichella, C.V. Alexandru, J. Shimagaki, C.A. Visaggio, G. Canfora, H.C. Gall, What would users change in my app? Summarizing app reviews for recommending software changes, in: Proc. Int'l Symposium on Foundations of Software Engineering (FSE), 2016, pp. 499–510.

[38] F. Martinez-Taboada, J.I. Redondo, Induction of decision trees, PLoS ONE (2020).

[39] C. Gerstenberger, D. Vogel, On the efficiency of gini's mean difference, Stat. Methods Appl. 24 (4) (2015) 569–596, https://doi.org/10.1007/s10260-015-0315-x.

[40] A. Trautsch, S. Herbold, J. Grabowski, Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction, in: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020, pp. 127–138.

[41] G. Canfora, A.D. Lucia, M.D. Penta, R. Oliveto, A. Panichella, S. Panichella, Multi-objective cross-project defect prediction, in: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, March 18-22, 2013, IEEE Computer Society, Luxembourg, Luxembourg, 2013, pp. 252–261.

[42] G. Grano, C. Laaber, A. Panichella, S. Panichella, Testing with fewer resources: an adaptive approach to performance-aware test case generation, IEEE Trans. Softw. Eng. 47 (11) (2021) 2332–2347, https://doi.org/10.1109/TSE.2019.2946773.