Graphical Abstract

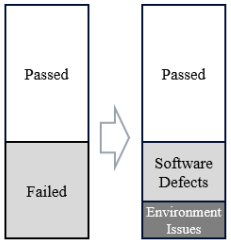
Predicting Test Failures Induced by Software Defects: A Lightweight Alternative to Software Defect Prediction and its Industrial Application

Lech Madeyski, Szymon Stradowski

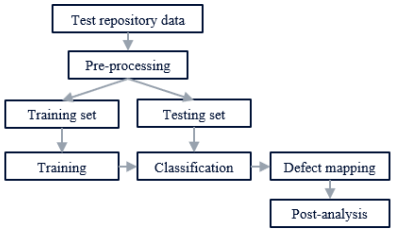
GOAL: A Software defect prediction solution that can work with system-level test process data and complement other defect prediction and test case selection and prioritization mechanisms in the company.

INPUT: Readily available system-level test repository data for NOKIA 5G.

ML SDP framework to predict test failures only related to software defects (without test and environmental issues). Using existing historical test failure data to software defect mapping.

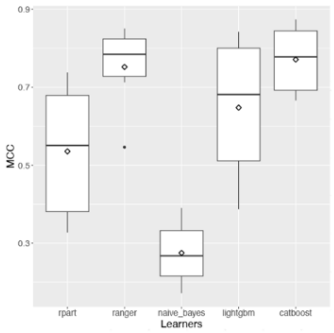


1. We use the existing test repository data base.
2. Filter out irrelevant test failures.
3. Use the proposed ML SDP framework to predict test failures only related to software defects (without test and environment issues).
4. Analyze the builds between test executions for software changes and mapping predicted test failures to past defects and fixed modules.

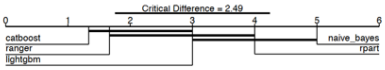


OUTPUT: Prediction of TEST RUNs that are likely to fail (due to software defects) that allows pinpointing faulty software modules without expensive test re-execution using the using the mapping of predicted test failures to past defects and corrected modules.

Boxplot for MCC across all tasks (data sets) using five times repeated 10-fold CV:



Critical difference diagram for MCC:



RESULTS: Good performance with MCC > 0.5

Highlights

Predicting Test Failures Induced by Software Defects: A Lightweight Alternative to Software Defect Prediction and its Industrial Application

Lech Madeyski, Szymon Stradowski

- We propose a Lightweight Alternative to Software Defect Prediction (LA2SDP)
- The idea behind LA2SDP is to predict test failures induced by software defects
- We use eXplainable AI to give feedback to stakeholders & initiate improvement actions
- We validate our proposed approach in a real-world Nokia 5G test process
- Our results show that LA2SDP is feasible in vivo using data available in Nokia 5G

Predicting Test Failures Induced by Software Defects: A Lightweight Alternative to Software Defect Prediction and its Industrial Application

Lech Madeyski[a,*], Szymon Stradowski[a,b]

[a] *Wrocław University of Science and Technology, Wyb. Wyspianskiego 27, Wrocław, 50-370, Dolnoslaskie, Poland*
[b] *Nokia, Szybowcowa 2, Wrocław, 54-206, Dolnoslaskie, Poland*

Abstract

Context: Despite a large amount of literature on Software Defect Prediction (SDP), its industrial applications are rarely reported and validated in vivo.
Objective: We aim to: 1) propose and develop a Lightweight Alternative to SDP (LA2SDP) that predicts test failures induced by software defects to allow pinpointing defective software modules thanks to available mapping of predicted test failures to past defects and corrected modules, 2) preliminary evaluate the proposed method in a real-world Nokia 5G scenario.
Method: We train machine learning models using test failures that come from confirmed software defects already available in the Nokia 5G environment. We implement LA2SDP using five purposely simple supervised ML algorithms and use eXplainable AI (XAI) to give feedback to stakeholders and initiate quality improvement actions.
Results: We have shown that LA2SDP is feasible in vivo using test failure to defect report mapping readily available within the Nokia 5G system-level test process, achieving good predictive performance. Specifically, CatBoost and Random Forest performed the best and achieved MCC $0.673 - 0.874$ in the repeated 10-fold cross-validation scenario and $0.815 - 0.820$ in the time-based scenario.
Conclusions: Our efforts have successfully defined and validated LA2SDP, enhancing the Nokia 5G system-level test process in vivo.

Keywords: software engineering, software testing, machine learning, software defect prediction, industry application, reproducible research, open science

*Corresponding author

1. Introduction

Machine learning software defect prediction (ML SDP or SDP for short) is a promising field of software engineering (SE). Its goal is to employ specific algorithms to analyse the product (e.g., a portion of software product metrics) or process (e.g., via process metrics) to estimate the number of defects in particular areas of the code (regression) or whether defects exist in those areas (classification) [1]. The business potential of such solutions is very high (see, e.g., [2, 3]). Still, the pace of industrial application lags behind academic research in the field, and there are very few publications on results obtained in vivo [4, 5].

Industrial applications of SDP are rarely reported due to a variety of reasons. For example, there is a lack of incentives for the industry to share real-world SDP experience, know-how or intellectual property. Furthermore, several researchers reported serious weaknesses in the SZZ (Śliwerski, Zimmermann, Zeller) algorithm [6] that is typically used in the SDP implementations [7, 8]. Namely, Herbold et al. [7] reported that only half of the bug-fixing commits determined by SZZ are actually bug-fixing (see further details in Section 3.2). This is a consequential problem when the SZZ algorithm is the core component of an in vivo SDP application. Last but not least, there is a need for extensive mining of software repositories to collect data just for the sake of SDP, which can be uneconomical from the company's perspective.

Thus, in this paper, we present and evaluate in vivo (in Nokia's system-level 5G test process environment) LA2SDP, a lightweight alternative to ML SDP. Notably, this work was inspired by the conclusions of a survey conducted by Stradowski and Madeyski [9], highlighting considerable opportunities to improve the quality and minimise the cost of software testing within the company. Furthermore, a study by Paterson et al. [10] states that defect prediction can accurately identify the modules that are most likely to be buggy In contrast, we aim to predict test cases that will detect software defects in particular modules to trigger post-analysis and potentially omit the costs related to retesting in expensive environments. Importantly, we have not encountered a similar approach that would study the possibility of analysing high-level test results from a test repository to be used for SDP instead of prioritising test cases ([11, 12]). Therefore, we are merging the aspects of test case selection and prioritisation (TSP) with software defect prediction (SDP) to open new avenues in software engineering research, address direct company expectations, as well as extend industry applications.

The presented research is part of a larger, business-driven effort to gather the challenges [9, 13], analyse existing methods [4, 5], and now develop a dedicated solution that satisfies the business requirements of Nokia. The most important contributions of this industrial study are highlighted below:

- Use case design of a software defect prediction solution that can work specifically with system-level test process data to complement other defect prediction and test case selection and prioritisation mechanisms within the company.
- Proposal of a lightweight alternative to Software Defect Prediction (LA2SDP) to satisfy the expectations (see Section 3.1).
- Industrial data suited for LA2SDP and benchmarked prediction models with code included in the reproduction package [14].
- Feature importance analysis to support interpreting and communicating the models to stakeholders and initiating improvement actions.
- Evaluation and discussion of the obtained results and practitioners' feedback.

In Section 2, we describe the background of our research and highlight its main contributions. In Section 3, we set the business context and describe the Nokia 5G test process. Next, Section 4 explains the methods used and the models that were built, followed by Section 5 that contains the analysis of the obtained results. Finally, in Sections 6 and 7 we present a discussion on the derived conclusions.

2. Related work

With our business-driven systematic literature review [5] we found several valuable primary studies that had big impact on our research efforts:

- Melo et al. [15] wrote a practical guide to support finding change-prone classes, which can help software professionals improve their product quality and steer future code changes. Furthermore, the authors apply the guideline to a case study based on a commercial data set. The approach consists of two phases: designing the data set and applying the prediction. In the first phase, it is necessary to choose the independent and dependent variables and collect the needed metrics. The application of the prediction step includes statistical analysis, normalisation, outlier detection, feature selection, resampling, cross-validation, tuning, selection of performance metrics, and ensuring reproducibility. In our study, we have followed a selected subset of these steps, described in detail in Section 4.
- Rana et al. [16] created a framework to support the adoption of ML SDP in the industry. Research highlights factors that need to be considered during in vivo introduction, such as general usefulness, reliability, and cost-effectiveness. The publication provides a comprehensive analysis of aspects rarely explored in academia, such as perceived barriers and benefits, availability of tool support, organisational characteristics, or needed competence ramp-up. The proposed framework influenced our research, especially in terms of new technology adoption challenges, building organisational readiness, and requirements definition.
- Furthermore, we internalised the experience report by Tantithamthavorn and Hassan [17] on defect modelling in practice, as it discusses several valuable recommendations, common pitfalls, and main challenges that were observed as practitioners attempted to develop SDP models in vivo. We have faced similar issues in our work, such as the risk of employing class rebalancing techniques when models are used to guide decisions, different learners providing greatly varied effectiveness, or replication difficulties due to confidentiality concerns.
- We compare our results with those of Malhotra and Sharma [18]. Their study uses 14 learners on Apache Click and Apache Rave data sets. It uses a filter-based correlation feature selection method to identify the most impacting predictors based on the area under the curve (AUC) performance measure. Next, Friedman-Nemenyi post hoc analysis is used to compare the results statistically. Although our study is based on test metrics and we do not apply as many learners, the overall methodology is similar and allows for indirect comparison.
- A second study to which we compare our results is the research by dos Santos and Figueiredo [19]. We use a similar methodology; however, in an industrial environment. The authors aimed to explore software features of ML SDP in a frequently used data set with five large open-source Java projects (Eclipse JDT and PDE, Equinox, Lucene, and Mylyn). Specifically, seven classification algorithms are evaluated using AUC and F1-score measures to select the best-performing learners.

To our knowledge, no secondary ML SDP studies focus on business applicability other than the work done by Stradowski and Madeyski [5]. This systematic literature review analyses publications on machine learning software defect prediction validated in vivo, where the authors identified 32 publications and documented relevant evidence of methods, features, frameworks, and data sets used in the industry. However, performed analysis also showed a minimal emphasis on feedback, practical lessons learned, and cost-consciousness within the reviewed publications, which are vital from a business perspective.

3. Project context

The challenges faced in the system-level testing of the 5G base station (called gNB or gNodeB [20]) at Nokia have been explored in the survey conducted by Stradowski and Madeyski [9]. Specifically, developing the 5G technology carries a considerable challenge. The difficulty arises from several factors, such as complex propagation characteristics needed for performance optimisation, wide frequency spectrum, hugely expensive over-the-air (OTA) test environments, complex verification of multiple-input multiple-output (MIMO) performance, defined by strict 3GPP specification requirements [20].

Consequently, the Nokia 5G gNB system is a grand and complex project, with over 60 million lines of code (LOC) written in C/C++ programming language. The gNB consists of tens of components and hundreds of modules developed by several major development organisations, each with hundreds of thousands of dedicated unit and integration tests. In later phases, the central software build with all integrated

deliveries is undergoing continuous testing in several globally distributed teams. Each team has its own set of requirements to validate the software against, as well as a dedicated laboratory infrastructure of varying complexity and associated maintenance costs. The test infrastructure can consist of servers with simulators and stubs, real Nokia gNBs in isolated setups, or massive anechoic chambers with multiple gNBs to measure signal interference and mobility scenarios. In our implementation project, LA2SDP is expected to support the existing effort in the system-level phase, being the last one of many test phases within the company [13], as it is the most expensive to operate and thus provides the most extensive opportunities for savings.

The Nokia test process follows the state-of-the-art and adheres to accepted standards, briefly introduced below. The company uses continuous delivery (CD) and continuous integration (CI) to build its products, emphasising left-shift principles[1] and strict phase containment[2] criteria. However, due to the complexity and size of the product, current phase containment rates are not satisfactory. Therefore, further quality improvement initiatives are necessary, and any advancement upon the current baseline brings consequential benefits as each escaped defect that was not detected in internal testing costs the company tens of thousands of dollars [9, 24]. The main three phases of the Nokia test process are described below and depicted in Figure 1:

- First, unit-level tests (UT) are under the responsibility of the development units (DU). Execution within the development units mainly happens in simulated environments and isolated hardware elements that integrate and validate basic functionalities before merging to the central software build.
- Second, entity testing (ET) is executed, after which an automated mechanism of software promotion builds central build packages where elements from all development units are integrated, and the whole system is made operational. This level is executed on a complete and real gNB system, allowing validation of its core functionalities.
- Last, a wide plethora of system-level testing (ST) such as continuous integration (CIT), continuous regression (CRT), and continuous delivery regression tests (CDRT) are executed. They contain a wide scope from new feature verification, stability, configurability, through security, to performance and capacity benchmarking.

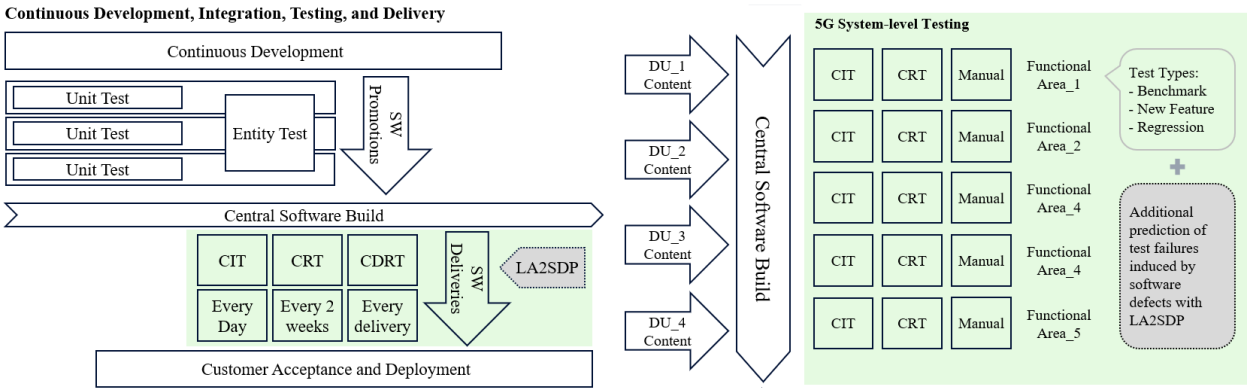


Figure 1: Nokia test process [9], with visualization of LA2SDP addition.

For the purpose of our case study, together with a group of five experienced test architects, we have run a feasibility study on the entire 5G gNB system-level environment (see Figure 1) in which the designed LA2SDP solution can be validated. System-level testing embraces regression, new features, and benchmark test case categories.

- Regression: By far the most numerous group of test cases, the purpose of which is to confirm that the legacy functionality works as it was designed and has not been broken by new code. As in many

[1]An approach where software testing is performed to find defects as early as possible in the life cycle, where they are cheaper to find and fix than the later stages [21, 22].

[2]Defect phase containment measures how many defects were caught before they escape into later phases [23].

software companies [25], the ever-increasing scope of maintenance testing in Nokia needs to be carefully managed.

- New Feature: All new content and resulting new functionalities must be tested to ensure they work as designed by the requirements. Such cases may range from a simple system reconfiguration to complex field verification scenarios requiring a close-to-real 5G network environment.
- Benchmark: As system performance is business critical for commercial wireless networks, extensive benchmarking needs to be executed to ensure that the added content increases the performance of the whole system. Such tests are challenging to execute, evaluate, and troubleshoot. Also, they usually take extended periods of time, lasting from one hour to several days, to thoroughly verify system stability. Hence, huge gains can be obtained by avoiding unnecessary tests with LA2SDP.

3.1. Expectations

For a successful implementation of a business-driven project, there needs to be a clear definition of the expectations that need to be satisfied [26, 27]. Hence, together with a group of five experienced test architects and quality managers from Nokia who participated in the project (see also Section 6.2), we defined the following set of high-level expectations for the final solution:

E1: The company wants to have a solution that is better than the average solution already reported by researchers. Shepperd et al. [28] found in their thorough analysis of ML SDP literature that the median and mean values of MCC are 0.305 and 0.308, respectively. The target was raised to MCC>0.5 to be on the safe side in case of fluctuating characteristics of the analysed projects/data sets. Auxiliary performance measures are also expected to be reported for convenience and transparency.

E2: We expect the LA2SDP solution to be as cost-efficient as possible and thus use already existing data and generally follow the KISS principle [29], which helps to keep the code, design, tests, and documentation fast and lean.

E3: We expect the ML models behind LA2SDP to be taken from the pool of models that support interpretability. It may be necessary during further steps of the ML SDP project [27].

Importantly, we do not aim to satisfy all expectations fully in this phase of research. Current efforts are aimed mainly at feasibility and verification of **E1** (for prediction effectiveness study see Section 4). Second, this research also provides baseline data for a dedicated study related to expectation **E2** (analysing cost-efficiency [24]) and enables possibilities for **E3** (interpretability of created models). However, at this project phase, it is considered enough to prefer the ML models supporting **E3** (e.g., [30, 31, 32]).

3.2. LA2SDP use case

The primary purpose of software testing is to discover defects and evaluate the quality of software artefacts. Testing can trigger failures that are caused by defects in the software and can directly find defects in the test object, but also result from environmental and process issues [33]. Test run results can be categorised into four types:

- True positive (TP): a test result where a defect is reported, and a defect actually exists in the test object.
- True negative (TN): a test result where a defect is not reported and no such defect actually exists in the test object.
- False positive (FP): a test result where a defect is reported although no such defect actually exists in the test object.
- False negative (FN): a test result which fails to identify the presence of a defect that is actually present in the test object.

Our system-level use case in Nokia complements an existing testing process within the company (Figure 1) where vast amounts of historical data exist and can be utilised in different ways. Specifically, we can filter the test run results to steer the modelling process towards different outcomes. For example, we can teach the ML

algorithms exclusively on failures with a confirmed cause identified, being software defects or environmental issues. Teaching the models on such subsets allows for predicting only test failures induced by software defects, which allows us to formulate LA2SDP. Alternatively, test run failures confirmed as environmental issues can be used to improve the test process itself, which allows us to formulate a complementary (to LA2SDP) method that we named the Lightweight Alternative to Test Process Improvement (LA2TPI).

Also, we can use the entire set to predict test case failure in general, i.e. Test Failure Prediction (TFP). That said, our current research effort focuses on the first LA2SDP option, predicting test case failures due to software defects, as this is the main requirement for the improvement proposal (see Section 3.1). Moreover, it also opens other research paths towards more effort-aware defect prediction [34, 35], as the number of predicted failures (and corresponding requirements) can help distinguish between modules with high and low defect density.

Obtained predictions that can be compared with the actual test execution results and detected discrepancies should cause additional post-analysis of a test case, leading to a discovery of a software defect in the associated code module. A matching result confirms the actual test outcome was correct. Consequently, an opposing result can trigger a test re-execution and post-analysis investigation to check for a false positive or false negative. Specifically, each discrepancy in the results has different implications:

- Real test passed, and the prediction shows the test case failed: real test should be repeated as a possible false negative, where a pass should be indicative of a false positive in the prediction, while if the repeated real test does not pass, additional post-analysis should lead to a discovery of a software defect in the associated code module.
- Real test failed, and the prediction shows the test case failed: this situation increases the probability of a software defect in the associated code module.
- Real test passed, and the prediction shows the test case passed: aligned results increase the probability of software defects not existing in the associated code module and lower the possibility of false positives.
- Real test failed, and the prediction shows the test case passed: real test should be repeated as a possible false positive, a pass should be indicative of possible issues with the environment or defective test automation procedure, indicating a lower chance of a software defect existing in the associated code module, while if the repeated real test does not pass, it increases the probability of a false negative in the prediction.

Analysing test history, which considers test cases that have recently failed, is closely related to defect prediction [10]. However, in our approach, the model is trained only on confirmed software defects and does not include environmental issues to make the gap even smaller. In our database, the failed test instances have been confirmed as defects by testers, who, after initial analysis, opened a defect report. Thus, there is considerable added value in using test case results leading to software defect predictions. Executing the test cases is hugely expensive, as some more sophisticated 5G test environments cost millions of EUR to build and maintain [9].

Test cases can also be time-consuming as specific stability scenarios take several hours to execute. Therefore, having an additional verification mechanism for defect prediction can bring considerable operational savings. Furthermore, based on the model results, we can temporarily omit low-risk areas not containing any defects, detect false positives limiting waste, or offer confirmation that discovered faults require a defect correction.

Also, we have compared our results with the defects that escaped to the customer in order to evaluate the coverage of the proposed solution. In our case, the failed test cases due to confirmed software defects at the studied system-level included more than 90% of all software defects discovered after integration. This way, we effectively (in more than 90%) address the SDP problem by solving the test failure prediction problem, which is easier to use as we avoid all of the widely discussed issues related to the SZZ algorithm [6] and its various implementations. The SZZ algorithm is the de facto standard for labelling bug-fixing commits and finding inducing changes for software defect prediction [7]. However, recent research uncovered many problems in different parts of this algorithm. For example, Herbold et al. [7] found that about one-quarter of the links to defects detected by SZZ are wrong due to missed and false positive links. Furthermore, they reported that due to the combination of wrong links and mislabelled issues, only about half of the commits

SZZ identifies are actually bug-fixing, and it misses about one-fifth of all bug-fixing commits. Herbold et al. [7] also confirmed the earlier results [36] that for every issue that is correctly labelled as a bug, there are 0.74 mislabelled bug issues. By employing LA2SDP, we escape from many problems related to the classic approach to SDP that require the SZZ algorithm and additional overhead related to calculating the necessary software metrics.

The tracking mechanism in Nokia was depicted in Table 1. A unique execution of a test case can pass or fail ('Test Run' and respective 'Test Result,' as in Table 2). After analysis and debugging, the failed test cases that end up as corrections in the software are marked with a resulting 'Defect Report'. Next, defect reports are tracked to specific areas of the code that have been corrected ('SW Module'), the team responsible for the modified code area ('Responsible Team'), the requirement of the product that this functionality satisfies ('Requirement'), and information about the failed 'Tested Build' (encoded as 'Build N'), as well as the build that this test has passed on last time it was tested 'Last Passing Build'. Furthermore, we can track all code changes ('SW changes') between the builds done in the area by the team ('Responsible Team') in a version control repository, which narrows down the changes that have been defective (changes between 'Build N' and 'Build N-y'). Consequently, putting all this information together allows tracking of a particular failed test run of a test case to defective changes (x' and y') in the code that has caused this failure. Such a mechanism can be applied to both real test case execution and machine learning predictions based on confirmed software defects. Furthermore, such a method can complement existing ML SDP approaches, making predictions on code and code change characteristics, by addressing the issue also from a black-box test perspective.

Test Case	**Requirement**	**Test Run**	**Test Result**	**Defect Report**	**Resp. Team**	**SW Module**	**Tested Build**	**Last Passing Build**	**SW Changes**
TC1	RQ1	Run1	PASS						
	RQ1	Run2	PASS						
	RQ1	Run3	FAIL	Report1	Team1	SWmodule1	Build N	Build N-x	x' changes
	RQ1	Run4	PASS						
TC2	RQ1	Run1	FAIL	Report2	Team2	SWmodule2	Build M	Build M-y	y' changes
	RQ1	Run2	PASS						
TC3	RQ2	Run1	PASS						
	RQ2	Run2	PASS						
TC4	RQ2	Run1	PASS						
TC5	RQ3	Run1	FAIL	Env. Issue	Filter out				

Table 1: Tracking mechanism used to connect failed test cases to software defects.

Identifying a failed test case does not equal finding a defect, but models trained on specific observations containing only confirmed software defects add considerable defect predictive value to the quality assurance process by offering a secondary verification mechanism to confirm or challenge test results. Since the test cases and open defect reports are tracked back to a specific product requirement, and each team owns a particular software area, we may assume that a predicted failed test case identifies a software defect in the given requirement.

This approach leads us to a new way, different from [37, 38, 39] that rely on build outcomes, to instantiate a lightweight alternative to SDP in industrial settings. Moreover, we are merging the aspects of test case selection and prioritisation (TSP) [40, 41] with ML SDP. According to Rothermel et al. [40], there are four objectives of TCP: increasing the detection rate at the beginning of the regression test execution, increasing the system code coverage under test, increasing high-risk fault detection rate, and increasing the probability of revealing faults related to specific code changes. Our efforts constitute a deep dive into the fourth aspect by translating aspects of TCP to direct software defect prediction outcomes. Hence, the LA2SDP process can be visualised as in Figure 2

Having outlined the context, company expectations, the specifics of the use case, we reiterate the terminology that will be used to describe the research details and outcomes:

- Test failure, as defined by International Software Testing Qualifications Board [33], is a deviation of

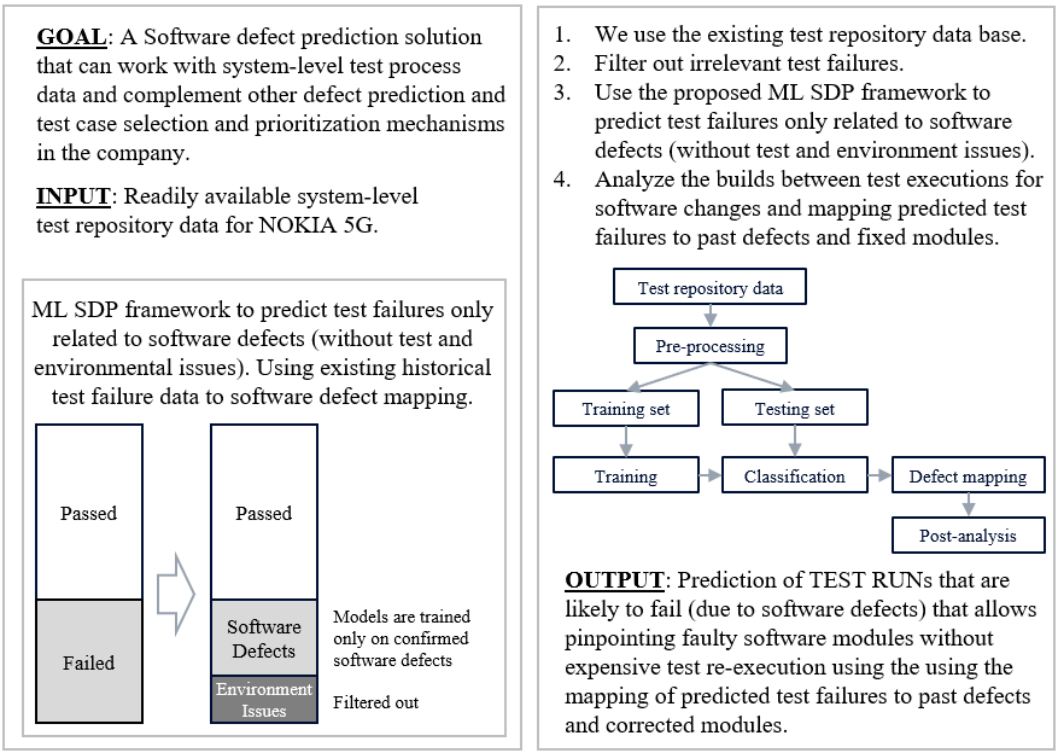


Figure 2: LA2DSP process visualisation.

the component or system from its expected delivery, service or result. A failure can be triggered by different reasons, such as defective software, hardware, or the test itself. However, we focus only on failures that were triggered by a software failure that is reported as a defect report and mapped in the test repository to the failed test run.

- Test data is a subset of process metrics that are an output of the testing process and are stored in a test repository. A wide plethora of features can be used, and ours are described in Section 3.2. Based on this data, we can predict test runs that will fail due to existing software defects, further using existing mapping mechanisms to impacted modules.
- Software defect prediction, is a predictive process of identifying modules that are defect prone. Accordingly, our aim is to predict test failures induced by software defects [42] and then to use existing mapping of past test failures to defect reports and impacted software modules to trigger a post-analysis and avoid expensive retesting. Hence, contrary to TSP, we do not aim to schedule test cases for execution in order to increase their effectiveness [40].

Hence, our alternative to established software defect prediction approaches by predicting test failures induced by software defects can be visualised as in Figure 3.

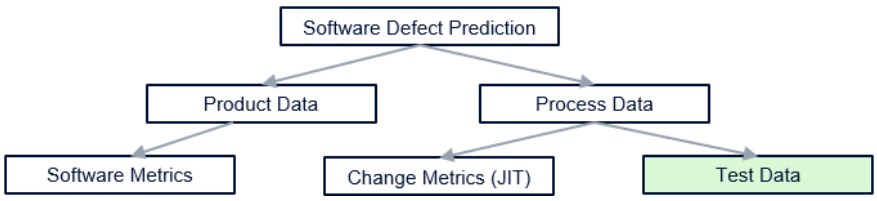


Figure 3: A simplified view to the software defect prediction approaches based on input types.

4. Methodology

We ask three research questions (RQ) in the initial phase of solution implementation:

RQ1: Can LA2SDP work with system-level test process data in Nokia 5G with expected performance (**E1**), assuming that we are allowed to use only already existing data (**E2**) and models that (according to literature) support interpretability (**E3**)[3]?

RQ2: Which learner employed in LA2SDP offers the highest performance in terms of MCC?

RQ3: What are the most important features in already existing data, and how can they be interpreted?

Also, we define success as satisfying the predefined requirements (see Section 3.1). Moreover, we collect feedback on the solution from subject matter experts in the company as an additional input source for validation and further improvement possibilities (see Section 6.2).

We started the data definition process by finding a suitable data set to satisfy our research questions. We gathered the data maintained in a dedicated test repository (called QC). We did not use the fault information stored in the fault report repository, as the QC data set was easier to obtain and analyse (which adheres to our second expectation **E2**). Also, as we investigated black-box system-level testing, we did not consider using software metrics [43]. The aforementioned decisions were made together with subject matter experts within Nokia and had a critical impact on the research and its outcomes (four experienced software engineers and two quality managers). When conducting research in a commercial environment with potentially important influence on many stakeholders, we wanted to make some initial inroads and show the value of ML SDP first, as then we would be better positioned to enhance the solution further based on the feedback and already the recognised expectation **E3**.

When considering the 5G software development life cycle (SDLC), we decided to focus on one of the interim stages of testing on the central build (see Figure 1).

Therefore, we only predict defects (that stem from commits) that can be detected at the system level, and at this point, we do not directly account for the escaped defects to the customer. Notably, the testing under scrutiny is black-box, meaning that the tester does not know the code implementation and focuses only on verification if the desired functionality works as specified, analysing only the input and output states.

4.1. Data set

The data set we use is a collection of historical test process metrics from the main test case repository for the Nokia 5G quality assurance process (see Table 2). Overall, it consists of almost 800,000 unique results for more than 100,000 test cases over a period of five and a half months from the beginning of January 2021 until the middle of June 2021, and executed into two-week feature builds. This constitutes a considerable database split by month into six separate files. This approach allowed us to compare the performance of particular learners and to analyse the differences between the performance of each learner on different data subsets.

The data was collected automatically from the central Nokia test repository and contains default features gathered for each test run entry by scripts and testers alike. The simple defect-code linking technique [44] is based on historical defect reports that are mapped in the test repository to previous failed test runs, and the relationship between bugs and modules is one-to-one. Here, a crucial consideration is the granularity of predictions on the test case, test instance, or test run level:

- A test case has the lowest granularity and can run on different builds and environments and by different teams. A test case is characterised by a requirement it satisfies and the author. Hence, in our context, providing test case-level predictions does not bring sufficient defect prediction value.
- A test instance is a test case assigned to a particular feature build, characterised by the tester, test line, and environment. Test instance, as an aggregation of test runs and traced to test cases assigned to a software build, is much easier for a human to understand with XAI, but it does not yet enable precise identification of the software module containing the defect.

[3] See Section 3.1 for expectations E1, E2, and E3.

- Last, the test run represents the highest granularity, as it reflects multiple repetitions of the same instance. Hence, it allows the detection of low-occurrence and utilisation of a wider range of features to predict which runs will fail.

Each granularity level can be tracked to past software defects and used for SDP; however, we have chosen the test run as it carries the largest amount of information in our context, as well as is already mapped to defect reports and software modules in the repository, avoiding any additional pre-processing actions.

Data set	**TEST. RUN.ID[1]**	**TEST. INSTANCE.ID[2]**	**EXECUTION. DATE[3]**	**TEST. PHASE[4]**	**RELEASE[5]**
QC1	74305	17871	31	61	10
QC2	144051	23137	28	62	10
QC3	145506	23722	31	65	12
QC4	149182	28321	30	72	11
QC5	138262	24957	31	70	11
QC6	64475	16281	30	62	10
Data set	**AUTOMATION. LEVEL[6]**	**TEST. OBJECT[7]**	**TEST. ENTITY[8]**	**ORGANI-SATION[9]**	**TEST. STATUS[10]**
QC1	2	3	3	127	2
QC2	2	3	3	138	2
QC3	2	3	3	135	2
QC4	2	3	3	157	2
QC5	2	3	3	161	2
QC6	2	3	3	157	2

[1] TEST.RUN.ID - Identification number of the test run.
[2] TEST.INSTANCE.ID - Identification of the test case.
[3] EXECUTION.DATE - date of TEST.RUN execution.
[4] PROGRAM.PHASE - name of the milestone for which the testing contributes.
[5] RELEASE - name of the software system release for which the testing contributes.
[6] AUTOMATION.LEVEL - TEST.AUTOMATION.LEVEL is information about whether the test run was automated or manual, and AUTOMATION.LEVEL.FINAL is the target state of the test automation.
[7] TEST.OBJECT - information if the test run was part of Benchmark, New Feature, or Regression.
[8] TEST.ENTITY - information if the test run was part of CIT (continuous integration testing), CRT (continuous regression testing), or Manual (see also Figure 1).
[9] ORGANISATION - name of the responsible organisation.
[10] TEST.STATUS - final state of the test run, and subject of our prediction (based on confirmed software defects).

Table 2: Data set statistics - count of elements and variable factors.

Software companies can possess immense amounts of information on their process efficiency. Although the mechanisms to obtain this information and their structure were not designed with machine learning tasks in mind, minimal pre-processing was needed.

There are differences between data sets, e.g., wrt. missing data in the 'TEST.ENTITY' field. Thus, we have attempted to understand the root cause, but such occurrences were occasional throughout all time frames (not only the six we can provide) and it was not related to data-gathering errors.

Such a situation reflects reality and needs to be considered in any research in industry. Second, there are no duplicate entries in the data sets, and each 'TEST.RUN.ID' represents a unique execution of a test case with unique characteristics (see Table 2). Last, each of our six data set files is heavily imbalanced, where number of passed test cases is much greater than the number of failed ones (less than 5%).

We train the models on data sets containing failed test cases confirmed by opened defect reports (see Section 3.2). Naturally, variables associated with failures (software defects and environmental issues) were removed from the data sets prior to the training phase. For data definition, each point consists of three parts [45], each described and adjusted to our context:

- Data element or sample: a particular execution of a test instance (test case assigned to a particular release), as the same test case can be run over different releases multiple times with varying outcomes.
- Variables or features: multiple characteristics describing each execution of the test run: ID, Instance, Date, Phase, Release, Automation, Type, Entity, Organisation, and Result, as presented in Table 2.
- Observation or label: in our case, the observation is a single test run execution result — pass or fail state. In reality, there are more states, such as blocked, postponed, ongoing, and similar inconclusive outcomes. However, we chose to focus only on the absolutes, as this is the primary goal of our solution.

Being the advocates of reproducible research in software engineering [46, 47, 48], we invite other researchers and practitioners to build upon the industry-relevant data [49, 50]. It would streamline the uptake of the presented ideas and solutions and encourage other researchers and practitioners to share their ideas for improvement and adaptations to specific characteristics of other companies and contexts.

Note: Data presented in the following sections have remained the same as the original values. However, to maintain confidentiality, we show the results obtained for historical test records (2021), not including the whole project repository but its selected subsets. Also, we had to remove potentially sensitive variables like test case titles or author names. Hence, the presented results can be perceived as underestimating results that could be achieved if the aforementioned actions were not taken.

4.2. Tools

We used the `mlr3` package in `R` language as a framework for predictive modelling (including also pre-processing) [51] and `DALEX` to support interpretability [52][4]. All code and models, as well as software version information about the OS, `R`, and loaded `R` packages are available in the Supplementary Material [14].

Note: All calculations were run on MacBook Pro Late 2023 (M3 Max, 48 GB RAM, Mac OS 14.3.1) and independently verified on Cluster Bem 2 provided by Wroclaw Centre for Networking and Supercomputing.

4.3. Learners

We have used the following supervised learners for our ML SDP modelling[5]

- Classification Tree (`rpart`) [53]
- Random Forest (`ranger`) [54]
- Naïve Bayes (`naive_bayes`) [55]
- Light Gradient-Boosting Machine (`lightgbm`) [56]
- CatBoost Gradient Boosting (`catboost`) [57]

The learners were chosen to fulfil the expectation **E3**, i.e., all used models support interpretability. Decision trees are very interpretable as shown by Molnar [32]. Naïve Bayes is also considered to be an interpretable model because of the independence assumption, as it is very clear for each feature how much it contributes towards a certain class prediction as we can interpret the conditional probability [32]. Likewise, the rest of the models also fulfil the expectation of interpretability according to the literature [30, 31].

Additionally, the selected models suit our data without major pre-processing or creating additional collection mechanisms not already available within the company (**E3**). The set of classifiers selected for benchmarking is considered enough to get prompt and early feedback on the achievable MCC level.

We planned to use 10-fold cross-validation (CV) resampling, as recommended in various studies, e.g., [58, 59], but to provide even more reliable results, we decided to use five times repeated 10-fold CV resampling, as well as time-based data set split where the QCdata_6 was used as the test set, while the earlier data sets (QCdata_1...QCdata_5) were used for training. In the case of five times repeated 10-fold CV the performance measures reported in Table 3 are averaged over the fifty runs (see further explanation of needed exact calculations [60] in related `mlr3` documentation[6]).

[4] See also `https://dalex.drwhy.ai/`.

[5] We will use the full names of the models, i.e., Classification Tree, Random Forest, Naïve Bayes, Light Gradient-Boosting Machine, CatBoost Gradient Boosting, and related short names, i.e., `rpart`, `ranger`, `naive_bayes`, `lightgbm`, and `catboost`, respectively, as they are named in the `mlr3` package used to build models) interchangeably in the paper.

[6] `https://mlr3.mlr-org.com/reference/resample.html`

4.4. Performance measures

Our data, as often is the case with real-world applications, are severely imbalanced. Hence, we need to use proper performance measure(s). There is ample research on the benefits and risks related to particular performance metrics in software defect prediction research [61, 62, 28, 63]. Taking into account the arguments and recommendations of Shepperd et al. [28], Yao et al. [63], as well as Chicco and Jurman [64], we decided to depend on the main comparisons and conclusions using Matthew's correlation coefficient MCC. Chicco and Jurman [64] concluded that MCC (that accounts for the whole confusion matrix, i.e., generates a high score in its $[-1; +1]$ interval only if the classifier scored high for all four quadrants of the confusion matrix) should replace the widely used ROC AUC as a standard statistic in all the scientific studies involving a binary classification. We decided to report the performance of our models with five additional auxiliary metrics only to enable comparison with other studies and give the readers and Nokia practitioners an even broader understanding of the obtained results.

5. Results

In this section, we analyse the results and conduct statistical analysis based on the MCC values. We run the global Friedman test with follow-up pairwise Nemenyi post hoc tests to verify whether the results are statistically different [65]. We comply with the recommendations by Kitchenham et al. [66] and used the probability of superiority $\hat{p}$ non-parametric effect size measures the probability of superiority ($\hat{p}$), as well as Cliff's delta (d). The range of $\hat{p}$ values is between 0 and 1, with 0.5 indicating stochastic equality and 1 indicating that the first group's results dominate those of the second, while Cliff's d ranges from -1 to 1, with 0 indicating stochastic equality, and 1 indicating that the first group's results dominate those of the second [67].

The averaged results (using five times repeated 10-fold CV) of each learner on each of the six data sets, as well as the results for the time-based split of the whole data set (QC_All) where the first five data sets were used for training, while the newest data set (QCdata_6) served as the test set, are shown in Table 3.

Importantly, such an approach constitutes a realistic scenario for the company. Next, we present the mean ranks over all data sets in Table 4. Our results show good modelling performance of our lightweight approach (see **E1** in Section 3.1), with the best models having MCC between 0.673 and 0.874 in the repeated 10-fold CV scenario, while in the time-based scenario the best models achieved between 0.815 (`ranger`) and 0.820 (`catboost`). Second, `catboost` and `ranger`, which is a fast implementation of Random Forest, were consistently among the two best classifiers, followed by `lightgbm`, `rpart`, and lastly `naive_bayes` (see also Figure 4, for the results of repeated cross-validation over all data sets, as well as Figure 5, for the results of the time-based split of the data set). Considering all data sets and all reported performance measures, `catboost` was the best in the case of MCC, ACC, Recall, and Fbeta (i.e., F-measure), while `ranger` was the best in the case of AUC and Precision (see Table 4). This is a testament to the importance of selecting a performance metric that fits the problem, as different results can be obtained depending on the chosen performance calculation method [63, 64]. MCC boxplots for each learner per data set (see Figure 6) show that while the results differ depending on the data set (despite being taken from the same project but in different time frames), the best-performing models, `catboost` and `ranger`, performed well on each data set.

Global Friedman statistical test allows us to conclude that MCC significantly differed between ML algorithms over all the tasks ($\chi^2 = 22.93$, $df = 4$, $p = 0.00013$), so we can proceed with the post hoc tests.

Further information on the MCC performance of the models can be visualised by critical difference diagram [65] (see Figure 7), where classifiers on the left are ranked the best, while those on the right are the worst. The thick horizontal lines connect learners that are not significantly different in ranked performance, so we may conclude that, based on MCC, `catboost` is significantly better than `naive_bayes` and `rpart`, but not significantly better than `ranger` and `lightgbm`, while `ranger` is significantly better than `naive_bayes`, but not significantly better than the others.

Pairwise comparisons of MCC using Nemenyi-Wilcoxon-Wilcox all-pairs test for a two-way balanced complete block design (see Table 5) tell us that `catboost` and `ranger` are significantly different from the `naive_bayes`. Comparing `catboost` with `naive_bayes` and `ranger` with `naive_bayes`, the post hoc test

task	learner	MCC	AUC	ACC	Recall	Prec.	Fbeta
QC_1	rpart	0.707	0.843	0.961	0.673	0.787	0.725
QC_1	ranger	0.834	0.993	0.978	0.758	0.941	0.840
QC_1	naive_bayes	0.315	0.843	0.817	0.649	0.240	0.350
QC_1	lightgbm	0.812	0.987	0.975	0.775	0.879	0.823
QC_1	catboost	**0.857**	0.991	0.981	0.832	0.905	0.867
QC_2	rpart	0.738	0.827	0.996	0.618	0.888	0.727
QC_2	ranger	0.850	0.995	0.998	0.767	0.946	0.847
QC_2	naive_bayes	0.338	0.946	0.965	0.754	0.161	0.265
QC_2	lightgbm	0.842	0.993	0.997	0.830	0.857	0.843
QC_2	catboost	**0.874**	0.994	0.998	0.817	0.938	0.872
QC_3	rpart	0.507	0.717	0.997	0.388	0.670	0.488
QC_3	ranger	**0.774**	0.990	0.999	0.608	0.988	0.751
QC_3	naive_bayes	0.221	0.953	0.972	0.665	0.078	0.139
QC_3	lightgbm	0.598	0.924	0.997	0.557	0.651	0.594
QC_3	catboost	0.750	0.987	0.998	0.625	0.904	0.737
QC_4	rpart	0.339	0.672	0.996	0.252	0.472	0.321
QC_4	ranger	**0.712**	0.986	0.998	0.526	0.971	0.680
QC_4	naive_bayes	0.172	0.941	0.959	0.628	0.052	0.095
QC_4	lightgbm	0.387	0.854	0.996	0.356	0.438	0.382
QC_4	catboost	0.666	0.985	0.998	0.554	0.810	0.653
QC_5	rpart	0.327	0.638	0.997	0.257	0.427	0.317
QC_5	ranger	0.546	0.989	0.998	0.392	0.773	0.515
QC_5	naive_bayes	0.214	0.959	0.964	0.811	0.060	0.112
QC_5	lightgbm	0.482	0.912	0.997	0.462	0.531	0.472
QC_5	catboost	**0.673**	0.986	0.998	0.561	0.815	0.660
QC_6	rpart	0.595	0.775	0.970	0.525	0.708	0.603
QC_6	ranger	0.795	0.991	0.984	0.690	0.934	0.793
QC_6	naive_bayes	0.391	0.853	0.916	0.633	0.289	0.397
QC_6	lightgbm	0.765	0.986	0.982	0.691	0.867	0.769
QC_6	catboost	**0.806**	0.987	0.985	0.747	0.887	0.811
QC_All	rpart	0.523	0.827	0.988	0.421	0.663	0.515
QC_All	ranger	0.815	0.997	0.995	0.712	0.938	0.809
QC_All	naive_bayes	0.200	0.848	0.900	0.594	0.087	0.152
QC_All	lightgbm	0.659	0.988	0.991	0.539	0.817	0.649
QC_All	catboost	**0.820**	0.992	0.995	0.764	0.885	0.820

Table 3: Averaged results for each learner on each data set QC_1...QC_6 (using five times repeated 10-fold CV) and the results for the time-based split of the whole data set (QC_All) where the first five data sets were used for training, while the last data set (QCdata_6) served as the test set.

yields a $p = 0.00057$ and $p = 0.00242$, respectively (see Table 5). Hence, in both cases, the difference is statistically significant, while non-parametric effect sizes $\hat{p}$ and Cliff's d are between 0.997 and 1.0, i.e., are considered large effect sizes according to [67]. Furthermore, the post hoc test yields a $p = 0.02875$ for the comparison of `catboost` with `rpart` (see Table 5), while effect sizes ($\hat{p} = 0.885$ and Cliff's $d = 0.770$) are considered large effect sizes as well [67].

The comparison of `catboost` with `ranger`, i.e., two best classifiers on our data sets, shows that the difference is not statistically significant with $p = 0.996$, while the effect sizes $\hat{p} = 0.543$ and Cliff's $d = 0.086$ are considered to be small effect sizes [67]. Hence, there is little to discern between both models; thus, both can be recommended.

For reference, we include exemplary benchmark results using the receiver operating characteristic (ROC) curve with confidence bounds for one of our data sets (QCdata_1), as well as ROC for the whole data set (QCdata_All where we used time-based split and QCdata_6 as the test set) in Figure 8, where we can observe the plotting of the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings for each classifier.

learner_id	MCC	AUC	ACC	Recall	Prec.	Fbeta
catboost	**1.333**	2.000	1.333	1.667	1.833	1.333
ranger	1.667	1.000	1.667	3.167	1.167	1.667
lightgbm	3.000	3.500	3.333	2.667	3.500	3.000
rpart	4.000	4.833	3.667	4.833	3.500	4.000
naive_bayes	5.000	3.667	5.000	2.667	5.000	5.000

Table 4: Mean rank of learners per measure

	rpart	ranger	naive_bayes	lightgbm
ranger	0.07872	-	-	-
naive_bayes	0.80900	0.00242	-	-
lightgbm	0.80900	0.58823	0.18304	-
catboost	0.02875	0.99621	0.00057	0.35857

Table 5: Pairwise comparisons of MCC using Nemenyi-Wilcoxon-Wilcox all-pairs test for a two-way balanced complete block design (p-value adjustment method: single-step).

More figures (also for other data sets) are available in the Supplementary Material [14].

We have also measured the time used for computation (on MacBook 2023, see Section 4.2), including both training and prediction duration, by each learner. Table 6 shows that `rpart` was the fastest, followed by `lightgbm`, `naive_bayes`, and then `ranger` and `catboost`. Thus, Light Gradient-Boosting (`lightgbm`) offers a good trade-off between speed and prediction performance, being less than two times slower than Classification Tree, while much faster than Random Forest (that was 39 times slower) and CatBoost (255 times slower), as the best-performing learners (`catboost` and `ranger`) needed longer computation times. Also, despite there is no statistically significant difference in predictive performance between those two learners, `ranger` was much faster, which might be an advantage in time-critical industry use cases.

learner_id	aggregated time [s]
rpart	1.58
lightgbm	2.82
naive_bayes	5.00
ranger	110.15
catboost	719.17

Table 6: The aggregated times (including train and predict phases) for each model are based on the statistics gathered by `mlr3`.

5.1. Enabling interpretability

To satisfy expectation **E3**, we exercise the opportunity to interpret/explain predictions on our real-world quality assurance process data from the very beginning of our adoption project [27]. Hence, we deployed an external, posthoc model-agnostic explainable AI framework `DALEX` that xrays a model and helps to explore and explain its behaviour. It was proposed by Biecek [52] as a dedicated `R` package to offer consistent methodology and tools for model-agnostic explanations, which create numerical and visual summaries. Also, it allows for comparing multiple models to help understand their relative performance, which we consider to do in subsequent phases of our research. Consequently, `DALEX` can explain the classifier for a specific single

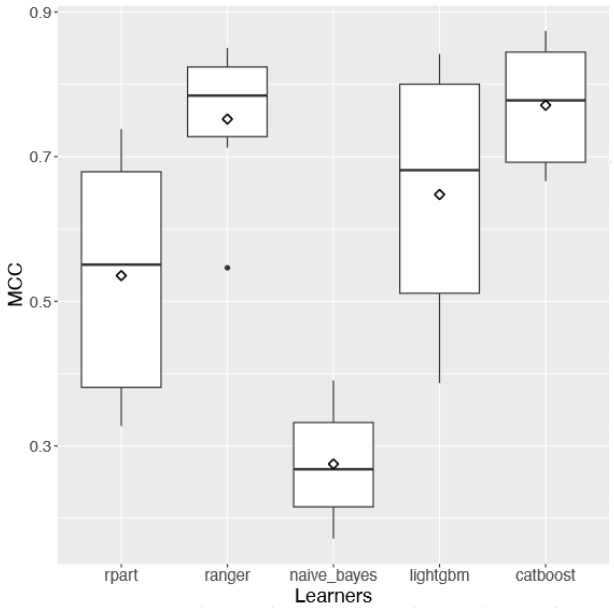


Figure 4: Boxplot (including mean marked as ◊) for MCC across all tasks (data sets) using five times repeated 10-fold CV.

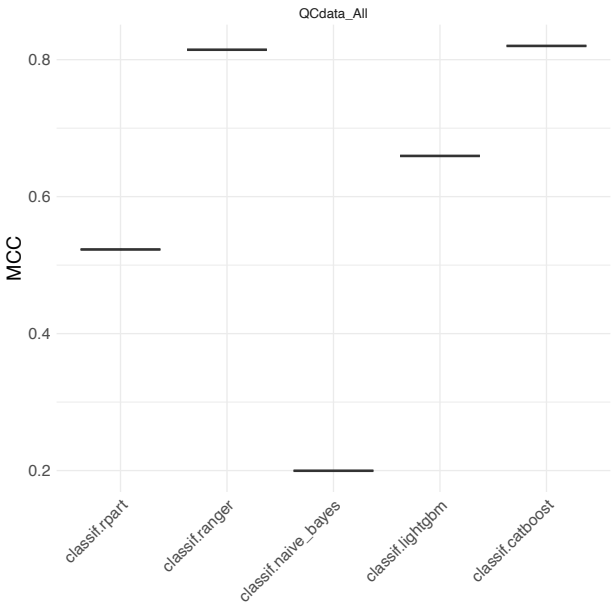


Figure 5: MCC for the time-based split where the first five data sets were used for training, while the last data set (QCdata_6) served as the test set.

instance of an already created model, enabling global and local understanding and is, therefore, suitable for our solution and sufficiently satisfies our goals (see Section 3.1).

Global model explanations (also called global feature importance) help us understand which features are most impacting in driving the predictions of the `catboost` and `ranger` models over the aggregated data sets. A convenient way to compute variable importance is to permute the features [54]. Accordingly, a feature is important if permuting the feature causes a large degradation in model performance. This approach can be applied to any kind of model (i.e., it is model agnostic), and results are simple to understand.

We illustrate the use of the permutation-based variable-importance evaluation by applying it to the `ranger` (RF) and the `catboost` model on the data set that is composed of all analysed data sets (see Figure 9). The dashed line in each panel of Figure 9 shows the loss[7] for the full model, being either the `ranger` (RF) or the `catboost`. Features farther to the right are more important as permuting them produces a higher loss (measured with $1 - MCC$). While EXECUTION.DATA, TEST.RUN.ID, TEST.INSTANCE.ID and ORGANISATION are the most important features for both best models, the order of importance differs[8]. Furthermore, the importance of the most important features decreases much less in the case of the `ranger` model than in the case of `catboost`.

The interpretations obtained through `DALEX` show that while feature importance varies between models (Figure 9), the test run, execution date, test instance, and organisation are the most impacting predictors. Meanwhile, the release, program phase, and automation levels are negligible. A detailed discussion of particular

[7]For the purpose of analysis of feature importance in the scenario where MCC is the performance measure of choice, it was necessary to implement a new loss function ($1 - MCC$) that is compatible with the chosen primary performance measure as `DALEX` does not provide the loss function based on MCC (if the performance measure of choice was AUC then `DALEX` offers the loss function $1 - AUC$). The implementation of the loss function (`loss_one_minus_mcc`) is included in the reproduction package in the R script `MadeyskiStradowski.R`.

[8]It is also worth mentioning that an optional decomposition of EXECUTION.DATE into YEAR, QUARTER, MONTH, WEEK, MDAY (day of the month), and WDAY (day of the week) had a negligible effect on MCC.

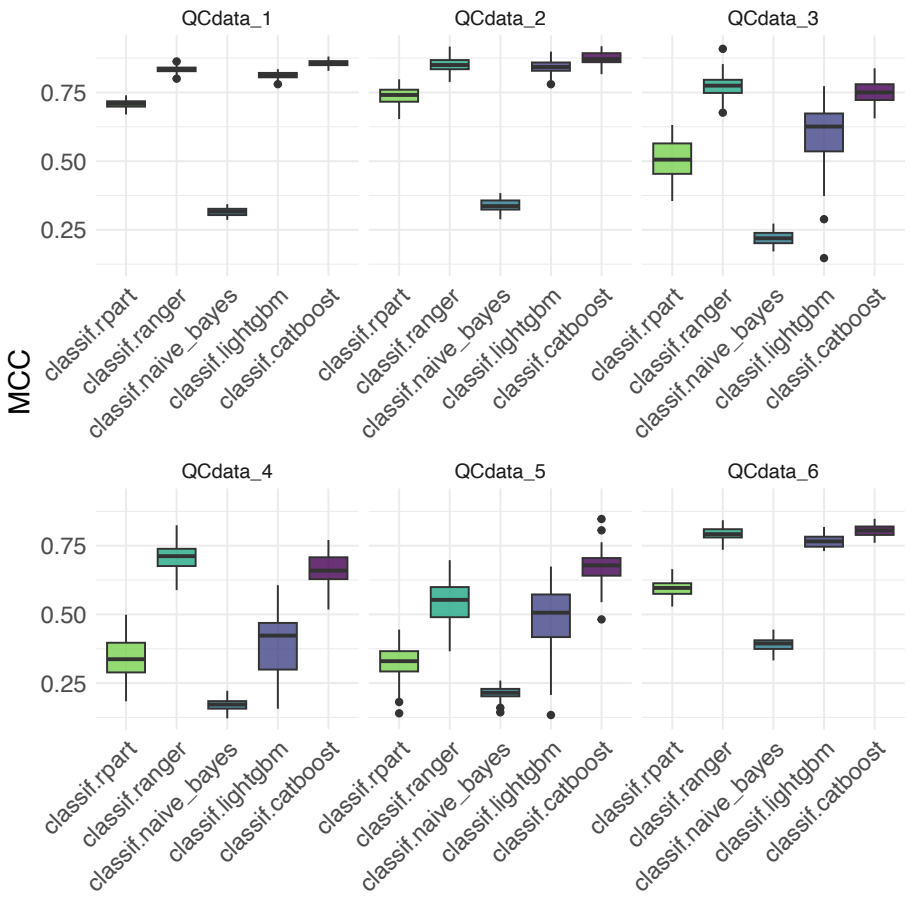


Figure 6: Benchmark results for each learner per data set using MCC boxplots.

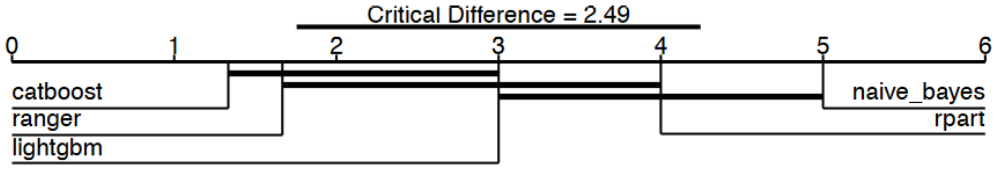
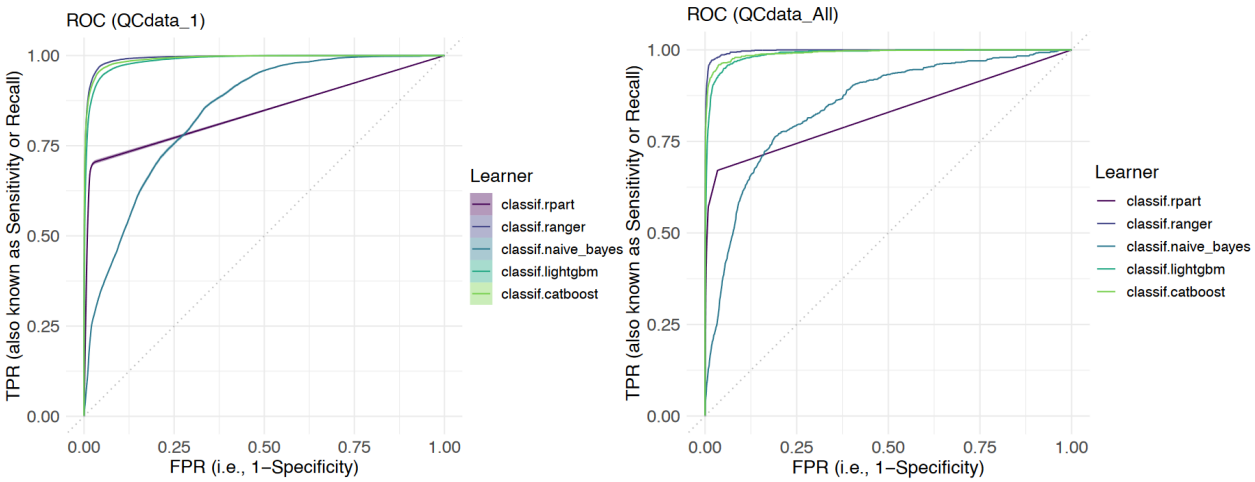


Figure 7: Critical difference diagram for MCC.

feature interpretations is presented in Section 6. Moreover, it is worth emphasising that the obtained results, backed by domain knowledge, can led to interesting improvement opportunities and new quality-oriented projects started in Nokia (see Section 6 for improvements **A1-A3**).

6. Discussion

The results show that the new approach is feasible and selected interpretable learners can achieve satisfying results with MCC much higher than the preliminary set threshold (see **E1**). Hence, the proposed lightweight solution (LA2SDP) fulfils the expectations posed in Section 3.1. However, due to the expectation **E2** to keep the solution as simple as possible, LA2SDP also has a high improvement potential that can be explored in subsequent phases of our research, namely:



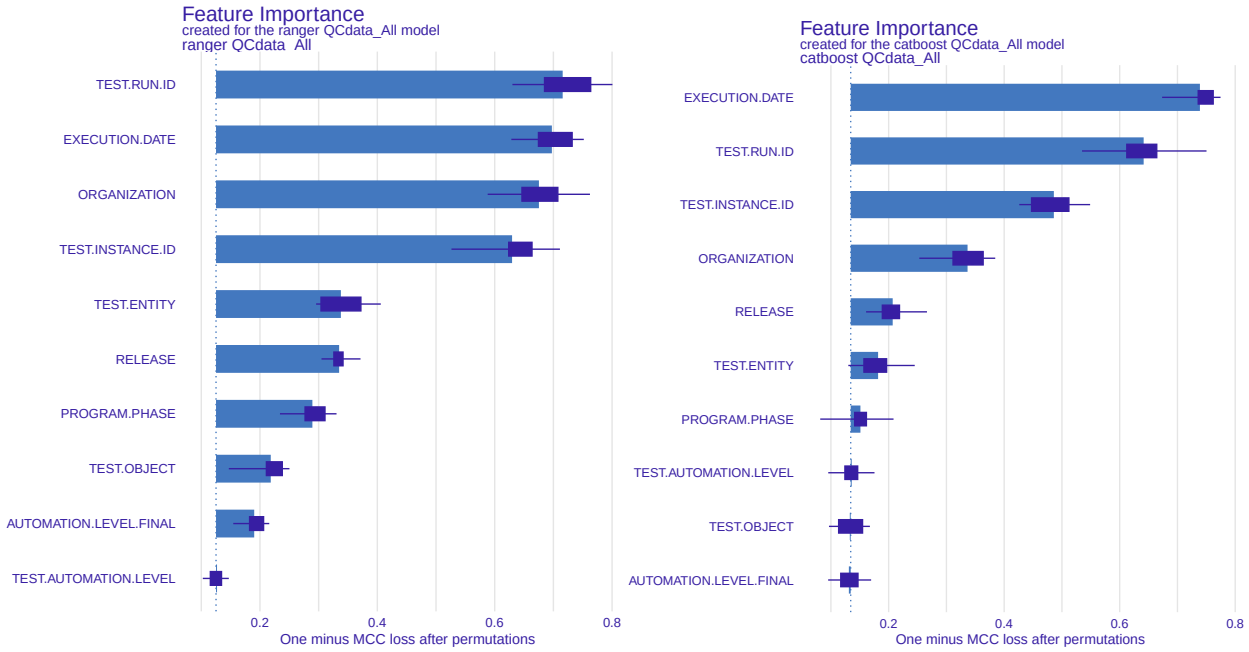
(a) 'QCdata_1'

(b) 'QCdata_All'

Figure 8: Benchmark ROC with confidence bounds for 'QCdata_1' and 'QCdata_All'

- We used relatively simple ML algorithms, and possibly better results can be obtained by employing more sophisticated models or pre-processing steps, as well as training models on larger data sets.
- Similarly, hyperparameter tuning was not performed and thus, optimisation with a more extensive computation budget can be used to improve performance further. We invite the academic community to achieve better results on our data sets as we consider doing ourselves in the future.
- We did not conduct an extensive study of feature extraction or selection [68, 69]. The features we could use due to confidentiality requirements were limited, and selecting the most contributing ones was not part of our success criteria. If the solution is to be commercialised and accepted as part of the standard process, feature selection (if not already supported internally by employed models) must be considered.
- Commercial data often suffer from missing samples. Our case is no different, mainly because the data set contains manually and automatically entered fields in the test repository. Missing entries have been omitted in calculations during the pre-processing phase; nevertheless, information carried by incomplete observations could have had predictive value [70, 71]. Hence, if the solution is accepted as part of the standard mode of operation, data imputation steps would be employed for new models that cannot account for missing values.
- Commercial data are frequently imbalanced. In each of our six data set files, the number of passed test cases is much greater than the number of failed ones (usually less than 5% of tests detect defects). Including mechanisms dealing with class imbalance is an absolute necessity. Class weight mechanisms were employed when supported by the models, but more sophisticated sampling algorithms can be considered if the LA2SDP solution is accepted as part of the standard mode of operation. For now, due to the class imbalance, our LA2SDP solution employs the MCC performance measure that considers all four quadrants of the confusion matrix, which is recommended in such an imbalanced scenario [28, 63, 64]. Also, MCC can later be used for hyperparameter optimisation.

The software development life cycle in Nokia 5G is a very complex process (see Section 3), and in consequence, the SDP life cycle is similarly challenging to manage. Thus, our project impacts a limited area within a larger ecosystem with numerous relationships, stakeholders, and requirements that must be thoroughly considered. That said, we have proposed an idea of how to manage such process complexity and how to embed SDP in multiple test phases by utilising the multidimensional knapsack problem [13], adhering to the company expectation that our solution works on system-level data complementing other already existing SDP and TSP mechanisms in the quality assurance process.



(a) The `ranger` (RF) model

(b) The `catboost` model

Figure 9: Feature Importance for the `ranger` (RF) and the `catboost` model

6.1. Answers to research questions

As suggested by Tosun et al. [72], it is important to decide how and when a defect prediction model will be used within the development life cycle. We aimed to understand if built models can be used with acceptable performance on one of the last phases of the SDLC. Therefore, our research sought a simple solution (**E2**) to gain value by proposing a lightweight alternative to SDP usable on industrial, real-world system-level test data sets. The proposed LA2SDP and obtained results showed that even relatively simple ML models can give adequate prediction results.

Despite using only five learners (Classification Tree, Random Forest, Naïve Bayes, Light Gradient-Boosting Machine, and CatBoost) for our lightweight predictive modelling, we achieved results that can be considered a solid starting point. The best models achieved satisfactory averaged results of MCC (0.673-0.874) on six data sets in the repeated CV scenario, and MCC (0.815-0.820) in the time-based scenario (with AUC close to 1), showing that the answer to our **RQ1** is positive.

> **Answer to RQ1** (Can LA2SDP be applied to the system-level testing of Nokia 5G with MCC>0.5 (E1) assuming that we are allowed to use only already existing data (E2) and models that (according to literature) support interpretability (E3)?): **LA2SDP can be applied to the system-level testing of Nokia 5G** (with all the imposed expectations **E1, E2, E3** fulfilled — achieving MCC>0.5, using existing data, and enabling interpretability).

To answer our **RQ2**, we wanted to understand which learner would offer the best performance in terms of MCC. We conducted statistical tests and calculated robust effect sizes to derive meaningful conclusions. Consequently, CatBoost and Random Forest turned out to be the best in terms of MCC, followed by Light Gradient Boosting, offering an interesting trade-off between speed and prediction performance. The results should not be unexpected as Random Forest is a learner widely known for its flexibility and providing excellent results most of the time, even without hyperparameter tuning [54, 73]. Random Forest combines the decision tree algorithm with bootstrap aggregation and constructs a large number of decision trees on

random subsets of the training data. It allows the algorithm to learn complex relationships between the features and the dependent variable [54]. Second, we found that CatBoost also offered great results without hyperparameter tuning. CatBoost is a relatively new Gradient Boosting algorithm developed at Yandex that offers an innovative approach for processing categorical features [57] (which appeared helpful due to the categorical features we operated with). On the other hand, it is essential to note that CatBoost and Random Forest were also the slowest in our comparison.

Answer to RQ2 (Which learner employed in LA2SDP offers the highest performance in terms of MCC?): **CatBoost Gradient Boosting (`catboost`) and Random Forest (`ranger`) were the best-performing learners in terms of MCC on all analysed data sets.** Statistical analysis shows that there is little to discern between both models; thus, both can be recommended for commercial implementation. Furthermore, `catboost` is significantly better than `naive_bayes` and `rpart`, but not `ranger` and `lightgbm`, while `ranger` is significantly better than `naive_bayes`, but not other classifiers.

In the described phase of implementation, we ensured, based on literature (see, e.g., [32, 30, 31]), that the models we used should support the interpretability of the results (see **E3** in Section 3.1). Furthermore, we successfully employed the `DALEX` package to demonstrate this ability in practice. That said, we focus more on this aspect in the subsequent phases of the adoption project [74].

Despite using various search engines and digital libraries, as well as conducting systematic mapping study [4] and systematic review [5], we did not find similar studies using test process information for comparison. However, we are able to discuss our results in the context of two studies that used similar prediction methods but were set in different contexts and used different inputs. First, we briefly compared our results with research by Malhotra and Sharma [18], despite the study being based on object-oriented metrics, whereas we use test repository data. Overall, in our case, the predictive capability using AUC was much higher than in the compared study, where the results averaged around 0.6 (the best results for different data sets were between 0.6-0.805), while in our case, it is as much as 0.92 on average and at least 0.986 for each data set in the case of the best models (see Table 3). The final ranking by Malhotra and Sharma [18] was also different, where Naïve Bayes turned out to be better than Random Forest.

We also compare our work to the research done by dos Santos and Figueiredo [19], which studies the software features impact on ML SDP performance using seven different classification algorithms and two measures on Java-based data sets. Apart from a custom-built Unbiased Search XGBoost algorithm, which turned out to be the best in all studies, Random Forest also proved very effective using both AUC and F1-score, similar to our case. Similarly, both research results demonstrate how a limited set of features can contribute to high AUC results.

Moreover, our approach is consistent with the conclusions of systematic literature reviews conducted by Durelli et al. [75], Pandey et al. [76], and Pachouly et al. [77] in terms of established state-of-the-art practices, and confirms the business potential of ML SDP. Importantly, our study also shows how relatively straightforward such implementation can be even in a complex industrial environment.

Finally, we have found answers to the last RQ when performing a feature importance study (Section 5.1). Next, we discussed the findings with our panel of Nokia experts to evaluate their significance, open opportunities for new domain expertise and knowledge discovery, and create options for starting dedicated improvement projects to increase product and process quality.

Answer to RQ3 (What are the most important features in already existing data, and how can they be interpreted?): The most important features are related to the **test run, execution date, test instance, and responsible organisation.** Furthermore, the discoveries brought **new domain knowledge and process improvement opportunities to the organisation.**

There are many benefits to be gained from building-in interpretability potential to the ML SDP solutions, such as transparency and trust, accountability and compliance, debugging and improvement, domain expertise,

and new knowledge discovery [78]. In our case, the main objective for enabling interpretability for the solution is the understanding requirement [79], as we want to use the results and explanations as an additional output for our users' benefit. Notably, one of the remaining reasons for the interpretability problem to remain unsolved is that interpretability is subjective and, therefore, challenging to measure and compare [79, 80]. Consequently, the usefulness of understanding predictions is domain- and context-specific, and it is necessary to consider the benefit of the use cases and the added value of each distinct feature importance insight.

Specific features and their interpretations are provided below (see also Figure 9) and have been discussed with the involved practitioners to design improvement actions (**A1-A3**):

- **TEST.RUN.ID** emerged as the highest influencing feature on the predictive performance, which came as a surprise to our participants. It is an internal counter for test runs in the test repository; however, after a deeper analysis of how the IDs are assigned, we now understand that it reflects all other essential features: date of creation and execution, test instance, and the responsible organisation. Therefore, despite the relative importance, we decided not to base any further action on this particular feature at this time.
- **EXECUTION.DATE**, meaning the date when the test run is executed, emerged as the second most important predictor. Furthermore, it has a significantly more impact than RELEASE and PROGRAM PHASE (result). This shows that continuous delivery and continuous integration (see also Section 3) work as designed, and the flow of defects is constant, rather than in bursts. This reflects the shorter cadence of new content being delivered to the central build based on two-week feature builds. On the other hand, there is a pattern of peaking defects at the beginning of each feature build entry when new functionalities are starting to be tested on the system level. This opened up a considerable opportunity to additionally steer the test schedules to be more optimal in terms of most risky tests executed first [23] (**A1**).
- **TEST.INSTANCE.ID** is a feature our engineering practitioners perceived as the most crucial in making software defect predictions. The test instance is a test (procedure description) prepared for actual execution, with a test assigned to a particular test environment and set required execution parameters (i.e., SW build, tester, external device). It contains all the most critical information for the engineer to identify the faulty SW module and gives meaningful inferences on the defect's location. Therefore, from a technical point of view, it is an essential feature for practitioners to enable software defect prediction mechanisms in our process, especially with the combination of organisation, test line, and granularity information (see also Section 5.1 for more information on the granularity of predictions). Importantly, further analysis of the interpretability information led to uncovering previously unknown relationships between defect-finding test instances and software module correction patterns. A dedicated project was started to investigate the usefulness of this information to further quality assurance improvements (**A2**).
- **ORGANISATION** turned to be one of the strongest characteristics, which our practitioners also expected as specific organisations are responsible for different code areas, and some modules are much more defect-prone than others [81]. Hence, test teams responsible for the more defect-prone SW modules fail more cases that are induced by software defects; however, failures due to test incorrectness of environmental issues (which we filtered out, Section 3.2) are comparable. Second, this is an essential consideration for the high-management stakeholders. ML SDP and XAI results should steer investments in the organisations' capacity to maximise defect finding and help prioritise the limited test resources on the most risky SW modules [82]. Furthermore, in our process, test engineers are usually responsible for dedicated test lines, and crosschecking of organisation and test instances can help understand why specific test lines cause defect reports much more frequently than others. Also, this is an important consideration for the particular organisation's management as it can steer investments in test infrastructure to maximise phase containment and efficient hardware utilisation. Namely, we found that a larger-than-expected number of software modules are executed for specific configurations, which led to starting another project to investigate the ramifications (**A3**).
- **TEST.AUTOMATION.LEVEL** and **AUTOMATION.LEVEL.FINAL** of the analysed test runs had no meaningful influence on the defect prediction effectiveness. Therefore, a high ratio of automated

test cases to manual test cases did not directly influence defect discovery in particular software modules; however, it adds essential benefits of test automation on the quality assurance processes in terms of reliability, cost, and speed [83].

6.2. Feedback

Feedback from experts plays a crucial role in establishing the validity of the approach within the company. After implementation and obtaining initial results, we organised a feedback session with the involved technical staff and management to elicit additional perceptions, expectations, and challenges for the created solution [84, 5]. As mentioned in Section 3.1, five experienced test architects and quality managers from Nokia who participated in the project were interviewed over a series of retrospective meetings. The most important points were the following:

- During the retrospective and post-validation, all involved practitioners unanimously saw the obtained predictions as useful (confirming the observations by Wan et al. [84]).
- As we trained our model only on test cases that fail due to confirmed software faults (including true negatives and false negatives, where a tester created a defect report that only after deeper analysis turned out not to be a real defect), experts saw value on comparison between predictions and actual test results for both cases (Section 3.2).
- Also, exploring the second group of failed test cases that were not confirmed as software defects to predict defects in the test environment (see Section 3.2) was discussed.
- A business case evaluation from a monetary perspective was done, showing a positive Return on Investment (ROI). Details have been described in a dedicated publication: Stradowski and Madeyski [24].
- The specific feature importance values were valuable to practitioners and led to further evaluation under the company's improvement framework. Therefore, not only did the obtained new insight lead to knowledge discovery, but it also triggered specific follow-up actions [74].
- The management perspective on how to stabilise and treat ML SDP as a standard practice is important to facilitate the adoption. Aspects such as competence development, communication, and innovation, but also effort estimation, maintenance costs, and technical debt [85], need to be studied further to accelerate the process.
- From the process perspective, making sure the predictions are accurate for incoming new data sets is imperative. Hence, the iterative nature of new data availability defines the need for time-based data set split and evaluation, being the go-to approach in the industry (see also the new process visualisation in Figure 1).

Consequently, after the commercialisation of the solution, the 5G quality assurance in Nokia is planned to include the ML SDP mechanism that is time-based split to run every two weeks on new data, after each feature build (Figure 1). The model predicts failed runs of a test instance,' which are tracked to respective requirements and software modules for additional analysis. Furthermore, the model can be compared with the actual test results of said test cases to detect false negatives and further tighten phase containment. Second, based on the confidence of the obtained predictions, test architects can make decisions on omitting specific test cases, which, due to the fact that they can be very expensive to run [24], can lead to meaningful operational savings in each feature build. In the next steps, dedicated product and process quality improvements will be added to increase the confidence and trust in the predictions as well as a decision to run the modelling more frequently based on a cost-benefit analysis [74].

6.3. Threats to validity

Construct validity reflects to what extent the studied operational measures represent what the researcher has in mind and what is investigated according to the research questions. While the general approach we use has been validated in many contexts, academic and real-world [5], there is a range of performance measures to choose from. We decided to report a range of them to provide a comprehensive view of the results and to allow easier comparisons with other studies. However, to make the answer to our RQs more robust, we based

our conclusions on the MCC metric recommended to use in imbalanced scenarios [28, 63, 64]. Secondly, as the predictors we use are generated automatically as well as manually by test engineers, they may not reflect the results perfectly. Also, as there are more test metrics gathered in the Nokia system-level test process than we considered in this study, other features, including dynamic ones (e.g., test site metrics), could be used in the next steps. We have also recognised a relevant construct improvement possibility; we do not consider test case or defect priorities, which is desired from a business standpoint.

External validity concerns the extent to which it is possible to generalise the findings. As the main focus of the study is a proprietary industrial data set and process, the generalisability of the results is limited. Second, we applied our method to only one project within the data set; we did not draw any conclusions about cross-project defect prediction potential [86], which still should be further verified. However, our findings and proposed approach should be relevant and valuable for any large-scale industrial system where similar test repository data can be gathered. Reliability is concerned with the extent to which the data and the analysis depend on the specific researchers. Notably, SDP studies are prone to researcher bias, as shown by Shepperd et al. [28]. Hence, we ensure that our results are reproducible by other researchers [46, 47, 48] by providing all details (data sets, code, and results) in the Supplementary Material [14].

For quantitative analysis, the counterpart to reliability is conclusion validity [87], which, in our case, is concerned with the relationship between the machine learning model and the chosen performance measure (MCC). We applied both time-based data set split, as well as five times repeated 10-fold cross-validation, instead of classic 10-fold cross-validation as recommended by Kohavi [58], Bowes et al. [59], to obtain reliable conclusions. We also conducted statistical tests and calculated robust non-parametric effect sizes following the guidelines by Kitchenham et al. [66, 67]. Unfortunately, we have not found any studies using black-box test metrics for SDP; therefore, comparing our results with other research conducted in a similar context is challenging. We analysed and published a data set that was slightly modified from the original for confidentiality's sake and ensured the modifications were random to limit the impact on the derived conclusions. Last, we paid much attention to keeping a high standard for communication and documentation [28, 46].

7. Conclusions and future work

Our paper addresses two appealing application prospects of ML SDP. First, we show how test repository data can be used for the detection of software defects by filtering the test results to teach the learners only on test cases that directly lead to defect discovery, besides the established approaches typically relying on software and process metrics. The proposed approach limits the need for expensive retesting of test cases and triggers a post-analysis directly based on the prediction results and past-defects mapping. Second, we utilise a lightweight application of ML SDP instead of a classic but more complex solution using an SZZ algorithm implementation and code mining tools to gather software product or process metrics [1]. Thus, we exercise simplicity in obtaining meaningful industry adoption inroads.

Consequently, the proposed lightweight alternative to SDP was successful, utilising the existing test process data to predict test failures induced by software defects (named LA2SDP). Not only have we obtained satisfying results on all test repository data sets from the black-box system-level testing of Nokia 5G gNB, but we have also found the process relatively straightforward to implement with the `mlr3` ML framework. We have used five supervised learners, a time-based data set split and five times repeated 10-fold cross-validation, and six performance measures to build our models. The CatBoost Gradient Boosting and Random Forest algorithms ranked the highest in our evaluations, followed by the Light Gradient-Boosting Machine (being also much faster than the aforementioned best algorithms), Classification Tree, and Naïve Bayes (the best models achieved satisfactory results of MCC between 0.673 and 0.874). Furthermore, we analysed the variable importance for our models, where the date, test instance, and organisation proved to be beneficial predictors from a business standpoint and triggered dedicated quality improvement actions.

In the subsequent steps, we plan to conduct more empirical studies on additional real-world time-split data sets from Nokia 5G system-level testing, as well as consider employing additional learners, hyperparameter optimisation, and feature selection/extraction to achieve even better performance. Nevertheless, the results we obtained were sufficient to decide that further research efforts will be executed within the company to adopt a similar approach as standard practice and use it commercially. Accordingly, we have shown that

LA2SDP (Lightweight Alternative to ML SDP) has the potential to improve the quality assurance processes within Nokia as well as other companies and large software projects that employ precise tracking of executed test cases to found software defects, as well as software modules that contained them.

CRediT statement

Lech Madeyski: Conceptualisation, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Resources, Software, Supervision, Validation, Writing – original draft, Writing – review & editing, Visualisation.
Szymon Stradowski: Conceptualisation, Data curation, Investigation, Methodology, Writing – original draft, Writing – review & editing, Visualisation.

Declaration of competing interest

This research was carried out in partnership with Nokia (with Szymon Stradowski as an employee).

Acknowledgement

This research was financed by the Polish Ministry of Education and Science 'Implementation Doctorate' program (ID: DWD/5/0178/2021). Calculations have been partly carried out using resources provided by Wroclaw Centre for Networking and Supercomputing (`http://wcss.pl`), grant No. 578.

Appendix A. Supplementary material

All research artefacts required to reproduce the results (code, results, and data sets from Nokia) are available in the Supplementary Material [14]. At the start of the R script, `MadeyskiStradowski.R`, we run `set.seed(123)` and use the R package `renv 1.0.3` to manage package versions. In the Supplementary Material, we also include our lockfile (`renv.lock`) recording metadata about every R package so that the computational environment can be re-installed on a new machine. Reproducibility is often problematic when parallelisation of computations is used, and we heavily use parallelisation to speed up computations. To overcome this issue and support reproducibility, we employed the `future` package that ensures that all workers receive the same pseudo-random number generator streams, independent of the number of workers[9].

References

[1] L. Madeyski, M. Jureczko, Which Process Metrics Can Significantly Improve Defect Prediction Models? An Empirical Study, Software Quality Journal 23 (2015) 393–422. doi:10.1007/s11219-014-9241-7.
[2] J. Hryszko, L. Madeyski, Assessment of the Software Defect Prediction Cost Effectiveness in an Industrial Project, in: L. Madeyski, M. Śmiałek, B. Hnatkowska, Z. Huzar (Eds.), Software Engineering: Challenges and Solutions, volume 504 of *Advances in Intelligent Systems and Computing*, Springer, 2017, pp. 77–90. doi:10.1007/978-3-319-43606-7_6.
[3] J. Hryszko, L. Madeyski, Cost effectiveness of software defect prediction in an industrial project, Foundations of Computing and Decision Sciences 43 (2018) 7–35. doi:10.1515/fcds-2018-0002.
[4] S. Stradowski, L. Madeyski, Machine learning in software defect prediction: A business-driven systematic mapping study, Information and Software Technology 155 (2023) 107128. doi:10.1016/j.infsof.2022.107128.
[5] S. Stradowski, L. Madeyski, Industrial applications of software defect prediction using machine learning: A business-driven systematic literature review, Information and Software Technology 159 (2023) 107192. doi:10.1016/j.infsof.2023.107192.
[6] J. Śliwerski, T. Zimmermann, A. Zeller, When do changes induce fixes?, SIGSOFT Softw. Eng. Notes 30 (2005) 1–5. doi:10.1145/1082983.1083147.
[7] S. Herbold, A. Trautsch, F. Trautsch, B. Ledel, Problems with SZZ and features: An empirical study of the state of practice of defect prediction data collection, Empirical Software Engineering 27 (2022) 42. doi:10.1007/s10664-021-10092-4.
[8] G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, R. Oliveto, A comprehensive evaluation of szz variants through a developer-informed oracle, Journal of Systems and Software 202 (2023) 111729. doi:10.1016/j.jss.2023.111729.

[9] `https://www.jottr.org/2020/09/22/push-for-statistical-sound-rng/`

[9] S. Stradowski, L. Madeyski, Exploring the challenges in software testing of the 5g system at nokia: A survey, Information and Software Technology 153 (2023) 107067. doi:10.1016/j.infsof.2022.107067.
[10] D. Paterson, J. Campos, R. Abreu, G. M. Kapfhammer, G. Fraser, P. McMinn, An empirical study on the use of defect prediction for test case prioritization, in: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), 2019, pp. 346–357. doi:10.1109/ICST.2019.00041.
[11] C. Catal, D. Mishra, Test case prioritization: a systematic mapping study, Software Quality Journal 21 (2013) 445–478. doi:10.1007/s11219-012-9181-z.
[12] R. Pan, M. Bagherzadeh, T. A. Ghaleb, L. Briand, Test case selection and prioritization using machine learning: A systematic literature review, Empirical Software Engineering 27 (2022). doi:10.1007/s10664-021-10066-6.
[13] S. Stradowski, L. Madeyski, Can we Knapsack Software Defect Prediction? Nokia 5G Case, in: IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings, 2023, pp. 365–369. doi:10.1109/ICSE-Companion58688.2023.00104.
[14] L. Madeyski, S. Stradowski, Supplementary Material for "Predicting Test Failures Induced by Software Defects: A Lightweight Alternative to Software Defect Prediction and its Industrial Application", 2023. URL: https://madeyski.e-informatyka.pl/download/MadeyskiStradowski24Supplement.pdf.
[15] C. S. Melo, M. Cruz, A. D. F. Martins, T. Matos, J. M. da Silva Monteiro Filho, J. C. Machado, A practical guide to support change-proneness prediction, in: International Conference on Enterprise Information Systems (ICEIS), 2019, pp. 269–276.
[16] R. Rana, M. Staron, J. Hansson, M. Nilsson, W. Meding, A framework for adoption of machine learning in industry for software defect prediction, in: 9th International Conference on Software Engineering and Applications (ICSOFT 2014), SciTePress, 2014, pp. 383–392. doi:10.5220/0005099303830392.
[17] C. Tantithamthavorn, A. E. Hassan, An experience report on defect modelling in practice: Pitfalls and challenges, in: 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP'18, Association for Computing Machinery, New York, NY, USA, 2018, p. 286–295. doi:10.1145/3183519.3183547.
[18] R. Malhotra, A. Sharma, Analyzing machine learning techniques for fault prediction using web applications, Journal of Information Processing Systems 14 (2018) 751–770. doi:10.3745/JIPS.04.0077.
[19] G. E. dos Santos, E. Figueiredo, Failure of one, fall of many: An exploratory study of software features for defect prediction, in: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2020, pp. 98–109. doi:10.1109/SCAM51674.2020.00016.
[20] The 3rd Generation Partnership Project, 3GPP REL15, 2021. URL: https://www.3gpp.org/release-15, accessed: 19.08.2023.
[21] International Organization for Standardization, ISO/IEC/IEEE 29119-1:2022 Software and systems engineering — Software testing, 2022. URL: https://www.iso.org/standard/81291.html, accessed: 20.12.2022.
[22] L.-O. Damm, L. Lundberg, C. Wohlin, Faults-slip-through—a concept for measuring the efficiency of the test process, Software Process: Improvement and Practice 11 (2006) 47–59. doi:10.1002/spip.253.
[23] International Software Testing Qualifications Board, Advanced level test manager (ctal-tm), 2023. Accessed: 02.09.2023.
[24] S. Stradowski, L. Madeyski, Costs and Benefits of Machine Learning Software Defect Prediction: Industrial Case Study, in: ESEC/FSE Companion '24: 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering Proceedings, 2024. doi:10.1145/3663529.3663831.
[25] V. Garousi, R. Özkan, A. B. Can, Multi-objective regression test selection in practice: An empirical study in the defense software industry, Information and Software Technology 103 (2018) 40–54. doi:10.1016/j.infsof.2018.06.007.
[26] IIBA, Babok: A Guide to the Business Analysis Body of Knowledge, t. 3, International Institute of Business Analysis, 2015.
[27] S. Stradowski, Madeyski, Bridging the Gap between Academia and Industry in Machine Learning Software Defect Prediction: Thirteen Considerations, in: 38th IEEE/ACM International Conference on Automated Software Engineering, 2023, pp. 1098–1110. doi:10.1109/ASE56229.2023.00026.
[28] M. Shepperd, D. Bowes, T. Hall, Researcher bias: The use of machine learning in software defect prediction, IEEE Transactions on Software Engineering 40 (2014) 603–616. doi:10.1109/TSE.2014.2322358.
[29] A. Stellman, J. Greene, Learning Agile: Understanding Scrum, XP, Lean, and Kanban, O'Reilly, 2014.
[30] M. Aria, C. Cuccurullo, A. Gnasso, A comparison among interpretative proposals for random forests, Machine Learning with Applications 6 (2021) 100094. doi:10.1016/j.mlwa.2021.100094.
[31] A. V. Konstantinov, L. V. Utkin, Interpretable machine learning with an ensemble of gradient boosting machines, Knowledge-Based Systems 222 (2021) 106993. doi:10.1016/j.knosys.2021.106993.
[32] C. Molnar, Interpretable Machine Learning: A Guide for Making Black Box Models Explainable, 2 ed., Lulu, 2023.
[33] International Software Testing Qualifications Board, Foundation level syllabus v4.0, 2023. Accessed: 02.09.2023.
[34] T. Mende, R. Koschke, Effort-aware defect prediction models, in: 2010 14th European Conference on Software Maintenance and Reengineering, 2010, pp. 107–116. doi:10.1109/CSMR.2010.18.
[35] X.-Y. Jing, H. Chen, B. Xu, Intelligent Software Defect Prediction, Springer Nature Singapore, 2024. doi:10.1007/978-981-99-2842-2.
[36] K. Herzig, S. Just, A. Zeller, It's not a bug, it's a feature: How misclassification impacts bug prediction, in: 2013 International Conference on Software Engineering, ICSE'13, IEEE Press, 2013, pp. 392–401. doi:10.5555/2486788.2486840.
[37] L. Madeyski, M. Kawalerowicz, Continuous Defect Prediction: The Idea and a Related Dataset, in: 14th International Conference on Mining Software Repositories (May 20-21, 2017. Buenos Aires, Argentina)., 2017, pp. 515–518. doi:10.1109/MSR.2017.46.
[38] M. Kawalerowicz, L. Madeyski, Continuous Build Outcome Prediction: A Small-N Experiment in Settings of a Real Software Project, in: H. Fujita, A. Selamat, J. C.-W. Lin, M. Ali (Eds.), IEA/AIE 2021. Advances and Trends in Artificial Intelligence.

From Theory to Practice, volume 12799 of *LNCS*, Springer, Cham, 2021, pp. 412–425. doi:10.1007/978-3-030-79463-7_35.
[39] M. Kawalerowicz, L. Madeyski, Continuous build outcome prediction: an experimental evaluation and acceptance modelling, Applied Intelligence 53 (2023) 8673–8692. doi:10.1007/s10489-023-04523-6.
[40] G. Rothermel, R. Untch, C. Chu, M. Harrold, Prioritizing test cases for regression testing, IEEE Transactions on Software Engineering 27 (2001) 929–948. doi:10.1109/32.962562.
[41] J. A. Prado Lima, S. R. Vergilio, Test case prioritization in continuous integration environments: A systematic mapping study, Information and Software Technology 121 (2020) 106268. doi:10.1016/j.infsof.2020.106268.
[42] IEEE, IEEE Standard Classification for Software Anomalies, IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993) (2010) 1–23. doi:10.1109/IEEESTD.2010.5399061.
[43] Z. Li, X.-Y. Jing, X. Zhu, Progress on approaches to software defect prediction, IET Software 12 (2018) 161–175. doi:10.1049/iet-sen.2017.0148.
[44] G. Mausa, T. Galinac Grbac, B. Dalbelo Bašić, A systematic data collection procedure for software defect prediction, Computer Science and Information Systems 13 (2016) 61–61. doi:10.2298/CSIS141228061M.
[45] S. Pradhan, V. Nanniyur, P. K. Vissapragada, On the defect prediction for large scale software systems – from defect density to machine learning, in: 2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS), 2020, pp. 374–381. doi:10.1109/QRS51102.2020.00056.
[46] L. Madeyski, B. Kitchenham, Would wider adoption of reproducible research be beneficial for empirical software engineering research?, Journal of Intelligent & Fuzzy Systems 32 (2017) 1509–1521. doi:10.3233/JIFS-169146.
[47] B. Kitchenham, L. Madeyski, P. Brereton, Meta-analysis for Families of Experiments in Software Engineering: A Systematic Review and Reproducibility and Validity Assessment, Empirical Software Engineering 25 (2020) 353–401. doi:10.1007/s10664-019-09747-0.
[48] T. Lewowski, L. Madeyski, How far are we from reproducible research on code smell detection? A systematic literature review, Information and Software Technology 144 (2022) 106783. doi:10.1016/j.infsof.2021.106783.
[49] L. Madeyski, T. Lewowski, MLCQ: Industry-Relevant Code Smell Data Set, in: Evaluation and Assessment in Software Engineering, EASE'20, ACM, New York, NY, USA, 2020, pp. 342–347. doi:10.1145/3383219.3383264.
[50] L. Madeyski, T. Lewowski, Detecting code smells using industry-relevant data, Information and Software Technology 155 (2023) 107112. doi:10.1016/j.infsof.2022.107112.
[51] M. Lang, M. Binder, J. Richter, P. Schratz, F. Pfisterer, S. Coors, Q. Au, G. Casalicchio, L. Kotthoff, B. Bischl, mlr3: A modern object-oriented machine learning framework in R, Journal of Open Source Software (2019). doi:10.21105/joss.01903.
[52] P. Biecek, DALEX: Explainers for complex predictive models in R, Journal of Machine Learning Research 19 (2018) 1–5.
[53] M. Grochtmann, K. Grimm, Classification trees for partition testing, Software Testing, Verification and Reliability 3 (1993) 63–82. doi:10.1002/stvr.4370030203.
[54] L. Breiman, Random forests, Machine Learning 45 (2001) 5–32. doi:10.1023/A:1010950718922.
[55] I. Rish, An empirical study of the naïve bayes classifier, IJCAI 2001 Work Empir Methods Artif Intell 3 (2001).
[56] 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, in: NIPS, Curran Associates Inc., 2017, p. 3149–3157.
[57] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, A. Gulin, Catboost: Unbiased boosting with categorical features, in: 32nd International Conference on Neural Information Processing Systems, volume 31 of *NIPS'18*, Curran Associates Inc., Red Hook, NY, USA, 2018, pp. 6639–6649. doi:10.5555/3327757.3327770.
[58] R. Kohavi, A study of cross-validation and bootstrap for accuracy estimation and model selection, in: 14th International Joint Conference on Artificial Intelligence - Volume 2, IJCAI'95, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995, pp. 1137–1143.
[59] D. Bowes, T. Hall, J. Petric, Software defect prediction: do different classifiers find the same defects?, Software Quality Journal 26 (2016) 525–552. doi:10.1007/s11219-016-9353-3.
[60] G. Forman, M. Scholz, Apples-to-apples in cross-validation studies: Pitfalls in classifier performance measurement, SIGKDD Explorations 12 (2010) 49–57. doi:10.1145/1882471.1882479.
[61] M. Sokolova, G. Lapalme, A systematic analysis of performance measures for classification tasks, Information Processing and Management 45 (2009) 427–437. doi:10.1016/j.ipm.2009.03.002.
[62] M. Rizwan, A. Nadeem, M. A. Sindhu, Analyses of classifier's performance measures used in software fault prediction studies, IEEE Access 7 (2019) 82764–82775. doi:10.1109/ACCESS.2019.2923821.
[63] J. Yao, M. Shepperd, The impact of using biased performance metrics on software defect prediction research, Information and Software Technology 139 (2021) 106664. doi:10.1016/j.infsof.2021.106664.
[64] D. Chicco, G. Jurman, The matthews correlation coefficient (mcc) should replace the roc auc as the standard metric for assessing binary classification, BioData Mining 16 (2023) 4. doi:10.1186/s13040-023-00322-4.
[65] J. Demšar, Statistical comparisons of classifiers over multiple data sets, Journal of Machine Learning Research 7 (2006) 1–30.
[66] B. Kitchenham, L. Madeyski, P. Brereton, Problems with Statistical Practice in Human-Centric Software Engineering Experiments, in: Evaluation and Assessment on Software Engineering, EASE '19, ACM, New York, NY, USA, 2019, pp. 134–143. doi:10.1145/3319008.3319009.
[67] B. Kitchenham, L. Madeyski, D. Budgen, J. Keung, P. Brereton, S. Charters, S. Gibbs, A. Pohthong, Robust Statistical Methods for Empirical Software Engineering, Empirical Software Engineering 22 (2017) 579–630. doi:10.1007/s10664-016-9437-5.
[68] I. H. Laradji, M. Alshayeb, L. Ghouti, Software defect prediction using ensemble learning on selected features, Information and Software Technology 58 (2015) 388–402. doi:10.1016/j.infsof.2014.07.005.

[69] A. Agrawal, T. Menzies, Is "better data" better than "better data miners"?: on the benefits of tuning smote for defect prediction, in: 40th International Conference on Software Engineering, 2018, pp. 1050–1061. doi:10.1145/3180155.3180197.
[70] S. Kim, H. Zhang, R. Wu, L. Gong, Dealing with noise in defect prediction, in: ICSE'11: 33rd International Conference on Software Engineering, 2011, pp. 481–490. doi:10.1145/1985793.1985859.
[71] M. Kakkar, S. Jain, A. Bansal, P. Grover, Evaluating missing values for software defect prediction, in: 2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon), 2019, pp. 30–34. doi:10.1109/COMITCon.2019.8862444.
[72] A. Tosun, A. Bener, R. Kale, Ai-based software defect predictors: Applications and benefits in a case study, AI Magazine 32 (2011) 57–68. doi:10.1609/aimag.v32i2.2348.
[73] W. Fu, T. Menzies, X. Shen, Tuning for software analytics: Is it really necessary?, Information and Software Technology 76 (2016) 135–146. doi:https://doi.org/10.1016/j.infsof.2016.04.017.
[74] S. Stradowski, L. Madeyski, Interpretability/Explainability applied to Machine Learning Software Defect Prediction: Industrial Perspective, 2024. URL: https://madeyski.e-informatyka.pl/download/StradowskiMadeyski24c.pdf, (in reviews).
[75] V. H. S. Durelli, R. S. Durelli, S. S. Borges, A. T. Endo, M. M. Eler, D. R. C. Dias, M. P. Guimarães, Machine learning applied to software testing: A systematic mapping study, IEEE Transactions on Reliability 68 (2019) 1189–1212. doi:10.1109/TR.2019.2892517.
[76] S. K. Pandey, R. B. Mishra, A. K. Tripathi, Machine learning based methods for software fault prediction: A survey, Expert Systems with Applications 172 (2021) 114595. doi:10.1016/j.eswa.2021.114595.
[77] J. Pachouly, S. Ahirrao, K. Kotecha, G. Selvachandran, A. Abraham, A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools, Engineering Applications of Artificial Intelligence 111 (2022) 104773. doi:10.1016/j.engappai.2022.104773.
[78] A. Barredo Arrieta, N. Díaz-Rodríguez, J. Del Ser, A. Bennetot, S. Tabik, A. Barbado, S. Garcia, S. Gil-Lopez, D. Molina, R. Benjamins, R. Chatila, F. Herrera, Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI, Information Fusion 58 (2020) 82–115.
[79] D. V. Carvalho, E. M. Pereira, J. S. Cardoso, Machine learning interpretability: A survey on methods and metrics, Electronics 8 (2019) 832. doi:10.3390/electronics8080832.
[80] A. H. Mohammadkhani, N. S. Bommi, M. Daboussi, O. Sabnis, C. Tantithamthavorn, H. Hemmati, A Systematic Literature Review of Explainable AI for Software Engineering, 2023. arXiv:2302.06065.
[81] N. Fenton, N. Ohlsson, Quantitative analysis of faults and failures in a complex software system, IEEE Transactions on Software Engineering 26 (2000) 797 – 814. doi:10.1109/32.879815.
[82] J. Jiarpakdee, C. K. Tantithamthavorn, J. Grundy, Practitioners' perceptions of the goals and visual explanations of defect prediction models, in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE/ACM, 2021, pp. 432–443. doi:10.1109/MSR52588.2021.00055.
[83] V. Garousi, M. V. Mäntylä, When and what to automate in software testing? a multi-vocal literature review, Information and Software Technology 76 (2016) 92–117. doi:10.1016/j.infsof.2016.04.015.
[84] Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin, X. Yang, Perceptions, expectations, and challenges in defect prediction, IEEE Transactions on Software Engineering 46 (2020) 1241–1266. doi:10.1109/TSE.2018.2877678.
[85] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, D. Dennison, Hidden technical debt in machine learning systems, in: C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, R. Garnett (Eds.), Advances in Neural Information Processing Systems, volume 28, Curran Associates, Inc., Norwich, England, 2015, pp. 1–9.
[86] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, B. Murphy, Cross-project defect prediction: A large scale experiment on data vs. domain vs. process, in: ESEC/FSE'09, ACM, New York, NY, USA, 2009, p. 91–100. doi:10.1145/1595696.1595713.
[87] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering: An Introduction, 2nd ed., Springer, Berlin Heidelberg, 2012.